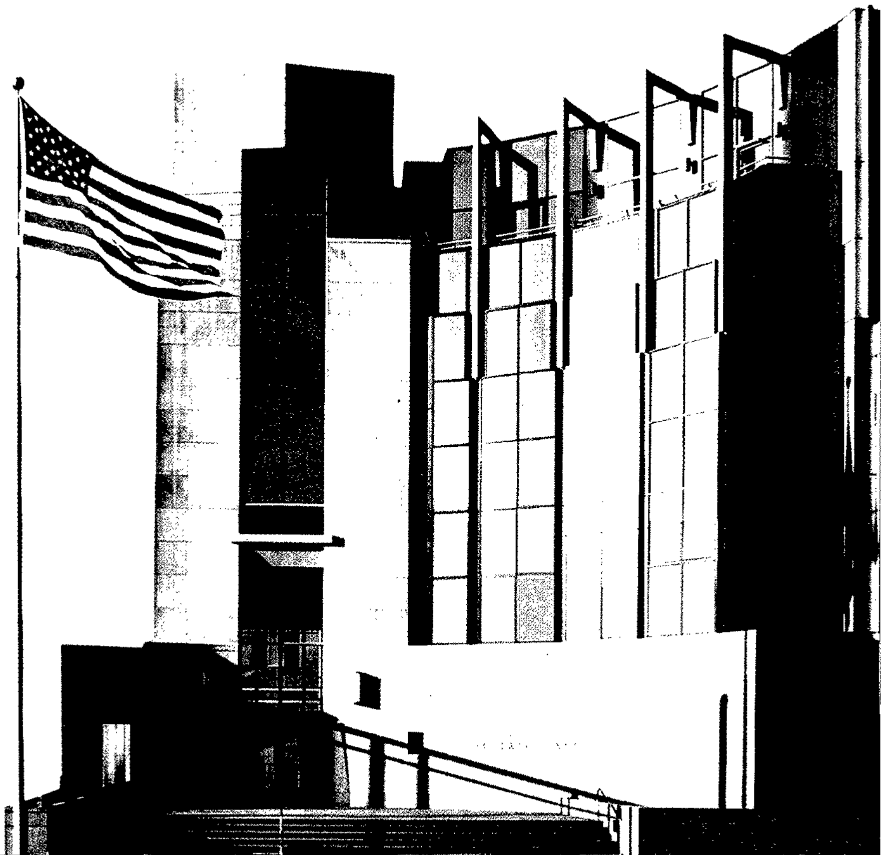
Carnegie Mellon
Software Engineering Institute

A Basis for Composition Language CL

James Ivers
Nishant Sinha
Kurt Wallnau

September 2002

Predictable Assembly from Certifiable Components Initiative



Technical Note
CMU/SEI-2002-TN-026

1122 108

A Basis for Composition Language CL

James Ivers
Nishant Sinha
Kurt Wallnau

September 2002

Predictable Assembly from Certifiable Components Initiative

Technical Note
CMU/SEI-2002-TN-026

Table of Contents

List of Figures

Abstract

CL is a composition language for predictable assembly from certifiable components. An application assembly process is predictable if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interaction. CL is similar to other composition languages that combine a component and connector style of description with a core compositional semantics specified in a process algebra. CL differs from these in its explicit treatment of details that are usually abstracted or ignored. For example, CL makes explicit the allocation of execution threads to component behavior; this distinguishes concurrent from sequential behavior, and leads to potentially smaller state spaces as well as more accurate behavioral descriptions.

This report describes the main concepts of CL and its rudimentary graphical syntax. This report also defines and illustrates the compositional semantics for CL using Hoare's CSP. The twin objectives of this report are to consolidate our current thinking about an ideal CL and to provide a starting point for the design of a practical and implementable CL. This report closes with a discussion of several open issues that must be resolved before this second objective can be satisfied.

1 Introduction

Software component technology emerged in the late 1990s as a promising means of simplifying the development of software applications. This simplification is achieved by imposing constraints, in the form of a *component model*, on component and application developers. A component model specifies a set of component types and rules for how different component types can interact. Component life-cycle management and other runtime services are provided by a component execution environment, often called a *container*. Component technology shifts the emphasis from developing new code to integrating or, more stylishly if not more accurately, *composing* existing software components in accordance with a component model.

Application development has indeed been simplified by component models and containers (in effect, *component standards*), but they apply almost exclusively to the syntactic or *constructive* aspects of composition, with the benefits typical of standard interfaces and interaction protocols. What has yet to be adequately addressed is the semantic or *behavioral* aspects of composition—how will a composite behave at runtime? Will it meet deadline, availability, safety, and security requirements?

1.1 Predictable Assembly from Certifiable Components

The Software Engineering Institute's (SEI's) Predictable Assembly from Certifiable Components (PACC) Initiative is addressing these behavioral aspects of composition. In this report, the term *assembly* will be used synonymously with the terms *composition* and *composite*. In our vocabulary, *predictable assembly* means that the *runtime* behavior of an assembly of components can be predicted from the properties of the components and their patterns of interaction. Further, components are *certifiable* if their properties can be ascertained and/or validated by disinterested third parties.

The following sections outline the key aspects of our overall approach to PACC and establish the context for the composition language CL.

1.1.1 Prediction-Enabled Component Technology

Our technical approach to PACC is to extend component technology with analysis technologies that support *compositional reasoning*, as entailed by our definition of predictable assembly. We call such an extension a *prediction-enabled component technology* (PECT). A PECT guarantees that assemblies are, by construction, predictable with respect to a selected set of runtime properties. Figure 1 depicts the fundamental idea of PECT and provides a framework to introduce PECT concepts and terminology.

The PECT component model defines assembly rules that are concerned exclusively with the constructive or syntactic aspects of composition; the Constructive Assembly in Figure 1 has

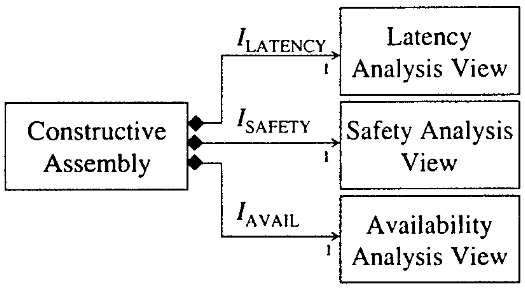


Figure 1: Interpretations Between Constructive and Analysis Views

been composed in accordance with this component model. Associated with each constructive assembly is one or more analysis views: latency, safety, and availability. Each analysis view can be thought of as input to a particular compositional reasoning technique. Analysis views are automatically derived from constructive views by means of a syntactic transformation, called an *interpretation*. Three interpretations are depicted in Figure 1: $I_{LATENCY}$, I_{SAFETY}, and I_{AVAIL}.

Interpretation may impose constraints on constructive assemblies beyond those found in the component model; these constraints take two forms. The first form is that specific component properties might be required. For example, $I_{LATENCY}$ might require that components have an associated property that describes their execution time or the scheduling priorities assigned to any independent threads within the component. The second form is topological constraints. Continuing with this example, $I_{LATENCY}$ might disallow any topology that does not enforce the priority ceiling protocol, that is, where the priority of a server is at least as high as the highest priority of any of its clients.

An interpretation is *consistent* if each constructive assembly is related to at most one analytic view under that interpretation. An interpretation is *valid* if there is empirical or formal justification for the claim that behaviors predicted in the analytic view will be manifested by the assembly of components specified in the constructive view.

1.1.2 CL: A Core Composition Language for PECT

The analogy between component models and programming languages is quite natural. In the former case, we are concerned with the well formedness of assemblies with respect to construction rules and one or more interpretations; in the latter case, we are concerned with the well formedness of programs with respect to a syntax and associated type theory. Each analysis view in Figure 1 reflects some theory about the runtime behavior of an assembly; no less is true of type systems. Type systems guarantee, by rules of syntax, the absence of certain classes of runtime behavior [Cardelli 97]. PECT guarantees, by rules of syntax, the analyzability of certain classes of runtime behavior.

The strength of this analogy and our own experience in building PECTs strongly motivate the need for a composition language, which we denote *CL*. CL gives formal meaning to a PECT component model—each constructive assembly is a sentence in the language of CL. The major syntactic elements of CL are *components* and *connectors*, something that CL has in common with most architecture description languages (ADLs). An annotation mechanism allows us to attach ⟨*property*, *value*⟩ pairs to components and connectors, and to finer-grained syntactic elements of CL. The values of these properties can be used to construct analysis-specific interpretations. Thus, CL is a core which can be extended with one or more analysis technologies, each of which imposes additional behavioral semantics on CL.

1.1.3 CL: Built-In Composition Semantics

One behavioral semantics is fundamental to the meaning of many interpretations and so is incorporated into the core CL. The semantics of component and interaction behavior—specifiable in a number of process algebras, notably CSP [Hoare 85], CCS [Milner 89], π-Calculus [Milner 99], and, more recently, FSP [Magee 01]—must be common to every interpretation.

While various ADLs [Magee 93, Allen 97] and self-described composition languages [Achermann 02] have also combined component and connector syntax with process algebraic semantics, CL differs by treating software components as implementations rather than as design abstractions. For example, CL preserves the asymmetry of the caller and callee; distinguishes asynchronous, synchronous, and reentrant behavior; and makes explicit the allocation of threads of control to component behavior. These and other "implementation details" often have a profound impact on assembly behavior.

These details also complicate composition semantics, that is, how assembly behavior is composed from component behavior specified in CSP. On the other hand, these details also offer opportunities to reduce the size of the composed state space, for example, by avoiding unnecessary parallel composition where concurrent behavior is impossible. This, we hope, will have positive ramifications on the tractability of model checking in PECT.

1.1.4 CL: Environment-Specific Connectors

In CL, the semantics of connectors is defined with respect to specific environment types. For example, an asynchronous message queue in a real-time environment might implement a small circular buffer with overwrite, while a non-real-time environment might implement (effectively) unbounded buffers with no overwrite.

In CL, the behavior of connectors is modeled separately from that of components, and connectors define the composition semantics of CL. Composition in CL always takes place in the context of an environment type, and each environment type defines a set of connectors that are valid within that environment type. Composition of assemblies across heterogeneous environment types is handled by explicitly modeling the shared environment type and the bridging connectors it defines.

Specifications of connectors are not trivial and can be simplified using specification patterns. One objective of this report is to produce a general approach for specifying the composition semantics for arbitrary interaction schemes (connectors) for arbitrary environment types E. We hasten to add here that end users of PECTs never see these semantic complexities any more than users of modern programming languages see the complexity of type checking or code generators.

1.1.5 CL: The Pin Style

We can think of each CL configuration, set of environment types, and set of interpretations as defining a distinct composition language or perhaps a distinct family of composition languages. Yet, despite the variation inherent in this scheme, each configuration shares with all others a single unifying constructive metaphor: the *Pin Style* or, more simply, *Pin.*

In Pin, components communicate exclusively through their pins: components receive stimulus through their *sink* pins and can stimulate other components through their *source* pins. No component can be its own stimulus, and only the environment can be an originating source of stimuli (another aspect of the definition of environment types). Therefore, Pin enforces a model of *pure composition*—an application is defined entirely in terms of a set of components and their connections. The behavior of a component is described in terms of *reactions* that specify the stimulus-response behavior of a component on its sink and source pins.

Connectors are used to link the source pins of one component to the sink pins of other components; these links represent the ability of components to interact. The semantics of connectors define how two or more component reactions are composed into a higher-order reaction. Two broad classes of interactions, and therefore connectors, are defined in Pin: asynchronous and synchronous. Beyond this, most of the semantic details of connections, for example their arity (e.g., 2-ary or N-ary connectors), are environment specific.

Before we proceed, we would like to warn the reader that the distinction between the Pin architectural style and the CL composition language is not always clear. The Pin style deals with concepts such as components, pins, their topologies, and so forth; CL is the formal syntax and semantics for component assemblies documented in the Pin style. CL is also the language in which the precise interaction semantics for different environment types are defined.

Determining which concepts belong in Pin and which ones belong in CL is not always an easy decision. For example, we believe that the distinction between synchronous and asynchronous pins is universal, so we consider that distinction part of Pin. However, the queueing semantics of asynchronous sink pins may differ among environment types and so should be part of CL.

Although we have attempted to make the proper distinctions between Pin and CL, we have not always been successful, primarily because we are still exploring the issue as we gain experience with defining the semantics of different environment types. When reading Section 3, keep in mind that the concepts described are part of Pin, while the syntax used in examples is part of CL.

1.2 About This Report

1.2.1 Objective of This Report

CL and its underlying Pin metaphor are under development; this report is a snapshot of our thoughts about them. Our intent is to both consolidate our understanding and to expose the main ideas to constructive criticism.

The primary focus of this report is on the specification and composition of component reactions over different types of connectors. Because the set of connector types and hence their semantics is open-ended, our long-term objective is to define specification patterns. In this way, we hope to reduce the complexity involved in developing PECTs and, perhaps, to automate some of the more tedious aspects of this development.

The secondary focus of this report is to establish preliminary ideas for a visual and textual language of CL. One premise of our approach to PECT is that, while the infrastructure of a PECT might be quite complex–comprising as it does different interaction schemes, interpretations, and behavioral models–much of this complexity can be "packaged" in high-level notations and tools. In this respect, CL and its supporting language environment requires serious attention in its own right.

1.2.2 Organization of This Report

We first provide an overview of related work in Section 2 and then introduce the major features of CL through a graphical syntax in Section 3. In Section 4, we define and illustrate a *naive* composition semantics for CL; we extend this definition to address a limited form of compositional minimization in Section 5. Examples of the naive semantics are provided in Section 6. The open issues that must be answered before CL can progress from its interim status are discussed briefly in Section 7. Finally, we close with a brief discussion of directions for further work in Section 8. In the appendix, we present a brief summary of the formal notations used in this report.

1.2.3 Audience for This Report

This report is targeted to computer scientists and software engineers interested in the formal specification of component-based software and in the use of these specifications for automated composition and compositional reasoning. Sections 2, 3, 7, and 8 require only general familiarity with component-based software technology, although some background in ADLs and coordination languages would be helpful. Sections 4-6, however, require the reader to have advanced knowledge of CSP [Hoare 85].

1.2.4 How to Read This Report

Readers interested in an overview of Pin and CL should read Section 3, the preliminaries of Section 4, and Section 7. Readers interested in comparing Pin and CL to other approaches should also include Section 2. Those readers interested in understanding our approach to compositional semantics should read the entire report.

2 Related Work

There has been substantial research under the general headings of module interconnection languages (MILs), architecture description languages (ADLs), coordination languages, and, more recently, composition languages. The extent to which these headings constitute distinct or overlapping areas of investigation is largely a matter of perspective. For example, one survey classifies Polylith [Purtillo 94] and Rapide [Luckham 95] under the heading of coordination languages, while their inventors originally classified them as MILs and ADLs, respectively [Papadopoulos 98].[1] It has even been argued that composition languages represent an amalgam of scripting, ADLs, and coordination languages [Achermann 02]. Despite some terminological confusion, the exemplars of these categories all seem similarly concerned with the search for language abstractions and semantics that address issues of complexity that are not conveniently addressed by the conventional imperative programming paradigm. What differs is only the type of complexity addressed: concurrency, distribution, mobility, and so forth. Here, we refer only to those exemplars that most closely resemble, or have most directly influenced, our thinking about CL.

Wright is an ADL best known for formalizing connectors as *first-class abstractions* [Allen 97]. Like Wright, CL distinguishes connectors from components and specifies the semantics of connectors in CSP. CL and Wright are also similar in their concern for representing system structure by making explicit the semantics of connectors and by exposing the interface signature of access points—*ports* in Wright, *pins* in CL. Unlike Wright, CL does not make connectors *first-class* elements of the language: there is no way in CL itself to define new types of connectors. Whether this makes CL less general than Wright is a long-running argument among ADL researchers. Darwin is another ADL with strong similarities to CL that, like Wright, has influenced the design of CL [Magee 93]. Darwin specifies component and assembly behavior using the FSP process algebra, which is more limited but substantially simpler than CSP [Magee 01]. Like CL, but unlike Wright, Darwin also makes structurally explicit the distinction between a component's incoming (*provides*) and outgoing (*requires*) interfaces. However, Darwin is more concerned with behavior than with system structure; unlike CL or Wright, it does not explicitly represent the boundaries and interfaces of deployed components. Koala is a Darwin-based development environment, in industrial use, for developing consumer electronics [van Ommering 02]. Like CL, it addresses implementation issues such as concurrency, but whereas concurrency is addressed in CL composition semantics, in Koala, it is addressed by introducing design primitives called *thread pumps* and *pump engines.* Tracta is a research prototype, also based on Darwin [Giannakopoulou 99]. Tracta is a toolkit environment for applying a variety of compositional verification techniques; it also implements a variety of compositional reduction techniques to ameliorate state space explosion under model checking. In many respects, we are aiming towards a Tracta-like environment with CL. We generalize from Tracta in two ways. First, the PECTs we construct using CL (perhaps more accurately, the Pin component model) include mechanisms to support empirical as well as formal analyses. Second, (although this is future work) we plan to "compile" CL into a standard intermediate representation such as

[1] Although the authors did distinguish MILs and ADLs in their survey, these categories were subsumed by the coordination languages category.

Bandera Intermediate Representation (BIR) [Corbett 99]; the intent is to simplify the integration of a variety of existing verification technologies, such as SMV [Cimatti 00], SPIN [Holzman 97], and FDR [Roscoe 98], into a PECT. In this respect, CL is similarly motivated to Acme [Garlan 97], an ADL interchange language. However, while Acme, like CL, defines a component and connector syntax and an annotation mechanism, it does not define a base composition semantics. PECOS is a component model for field devices [Nierstrasz 02]. Like Wright and CL, it defines both components and connectors. Like CL and Acme, PECOS provides an annotation mechanism; in all three, the annotations are used to provide analysis and tool-specific information. PECOS also specifies component behavior formally, although it uses petri nets in preference to process algebra. PECOS does not yet make use of petri nets for behavior analysis, although there are plans to apply it to timing analysis. Should these plans come to fruition, PECOS would satisfy our definition of PECT. Further, PECOS is specialized to memory-constrained devices and has, for example, a built-in cyclic scheduler. Last, Piccola is a composition language that combines aspects of ADL and scripting [Achermann 02]. Like CL and Wright, Piccola uses a process algebra to specify component and assembly behavior. However, instead of CSP, it uses $\pi\mathcal{L}$-calculus, a form of Milner's π-calculus, that provides a polymorphic extension to π's fixed positional, tuple-based interfaces [Milner 99]; this adds considerable expressive power to the language at the cost of some increased complexity. Piccola is also more general than CL in that it is not restricted to pure composition; the scripting language embeds considerable application semantics, as does a glue language that is used to adapt components to remove mismatches among their interfaces or behaviors. It is arguable whether this more general focus is laudable or necessary.

3 Overview of Pin

3.1 Notational Conventions

In the following discussion, entities are denoted by their names. We use component names c_i, sink pin names s_j (for the jth *stimulus*), and source pin names r_k (for the jth *response*), where $i, j, k \geq 0$. Where the distinction between sink and source pin is irrelevant, we will use pin names p_k (for pin_k). In all cases, we omit subscripts where they are not required. Components define a scope for pin names; within the scope of a component, all pin names must be unique. The expression $c.p$ denotes pin p of component c. We use capitalized names to denote predicates defined on pins. For example, $SomeCondition(c.p)$ is *True* if pin $c.p$ satisfies *SomeCondition* and is *False* otherwise. We use lowercase names to denote properties of pins. For example, $someProperty(c.p)$ denotes the value of property *someProperty* of pin $c.p$.

3.2 Components and Pins

A component interacts with its environment exclusively through its pins; there are no other communication paths to or from a component. As mentioned in 1.1.5, there are two types of pins: *sink* pins and *source* pins. A component receives communication (stimuli) on its sink pins and initiates communication (responses) on its source pins. Figure 2 depicts the graphic notation we use and refer to later. By convention, we put sink pins on the left side of a component and source pins on the right side. This allows us, at the cost of some cultural bias, to "read" the component, and sometimes their assemblies, from left to right.

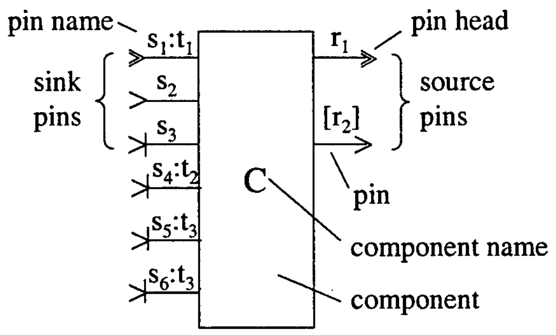


Figure 2: Graphical Notation for a Component and Its Pins

3.3 Pin Data Interface

Each pin has a *data interface*, denoted by $interface(c.p)$. For some $c.p$, $interface(c.p)$ describes the type $\langle T_0 \times T_1 \times \cdots \times T_n \rangle$, where each T_j is the primitive data type transmitted on the pin and each primitive type is one of

$\{SByte, UByte, SWord, UWord, SDWord, UDWord, SDouble, Float, String\}$. A complete specification of CL for a particular environment would specify the representation of these types; we do not do so here.

As in conventional interface specification, each data type corresponds to a formal argument that has a parameter-passing mode, where mode is one of $\{In, Out, InOut\}$. Note that a pin's data type interface might be specified explicitly or be inferred from a CSP specification. This issue remains open.

3.4 Source Pins

There are two types of source pin–asynchronous and synchronous–that satisfy the predicates $Asynchronous(c.p)$ and $Synchronous(c.p)$, respectively.[2] Source pins represent the ability of a component to initiate, in response to some original stimulus, an interaction with other components.

It is reasonable to think of asynchronous source pins as message sends and synchronous source pins as procedure calls, but too much faith should not be put on this gross interpretation, as we have not always implemented asynchronous connectors in this way. Nonetheless, arguments in an asynchronous pin data interface are restricted to mode *In*. Source pins are graphically denoted with pin heads $\gg$ if $Asynchronous(c.r)$ and $>$ if $Synchronous(c.r)$. In Figure 2, $c.r_1$ is an asynchronous source pin, and $c.r_2$ is a synchronous source pin.

If $Optional(c.r)$, then $c.r$ is an optional source pin, otherwise it is mandatory. Mandatory source pins must be connected in an assembly, while optional source pins need not be connected. This models the distinction between calls (optional) and uses (mandatory), respectively. A component c_0 uses sink pin $c_1.s$ if the behavior of c_0 depends on the correct behavior of $c_1.s$; otherwise it only calls $c_1.s$. Graphically, we distinguish optional source pins from mandatory ones by enclosing the names of optional sources in braces [ ]. So, $c.[r_2]$ is an optional source pin in Figure 2.

3.5 Sink Pins

There are two types of sink pins, asynchronous and synchronous. Analogously with source pins, they represent the ability of a component to receive asynchronous or synchronous stimulus, respectively. As with source pins, the data type interface of an asynchronous sink pin is restricted to arguments of mode *In*. Sink pins are graphically denoted with pin heads $\gg$ if $Asynchronous(c.s)$ and $>$ if $Synchronous(c.s)$. In Figure 2, $c.s_1$ is asynchronous, while $\{c.s_k \mid 2 \leq k \leq 6\}$ are synchronous.

If $Threaded(c.s)$, $c.s$ has its own thread of control, and $threadId(c.s)$ denotes its identity. Threads represent units of concurrent execution and may be implemented by operating system threads, processes, tasks, and so forth. Graphically, $t_j = threadId(c.s)$ is denoted as a suffix $: t_j$ on the sink pin name. In Figure 2, sink pins $c.s_2$ and $c.s_3$ are unthreaded, but $c.s_5$

[2] Currently, $Asynchronous(x) \Leftrightarrow \neg Synchronous(x)$, but it seems more flexible, if verbose, to define two predicates rather than just one. This is true of other predicates as well.

has the thread $threadId(c.s_5) = t_3$. Threads may be shared by sink pins. So, in Figure 2, $c.s_5$ and $c.s_6$ share thread t_3, but $c.s_1$ and $c.s_4$ have their own threads. Note that asynchronous sink pins must be threaded, that is, $Asynchronous(c.s) \Rightarrow Threaded(c.s)$.

If $Mutex(c.s)$, $c.s$ is called a mutex sink, and only one caller may be active on $c.s$ at any given time; in effect, the caller must obtain the semaphore for $c.s$. Conversely, if $\neg Mutex(c.s)$, the $c.s$ is called a reentrant sink and is never guarded by a semaphore. Note that even a reentrant $c.s$ might force a caller to wait while it synchronizes on an internal (to $c.s$) resource. We obviously care about this property only for synchronous sink pins, since asynchronous callers do not wait and cannot be blocked. Graphically, mutex sinks are represented with the pin head >|. In Figure 2, $Mutex(c.s_j), 3 \leq k \leq 6$. Note that $Threaded(c.s) \wedge \neg Asynchronous(c.s) \Rightarrow Mutex(c.s)$.

3.6 Assemblies and Environments

An *assembly* is defined as a set of components and their connections. We denote an assembly as a_j. An assembly is graphically depicted as a box enclosing a set of connected components; this visually reinforces the notion of an assembly as a scope of, or container for, components. Thus, $a.c$ denotes component c in the scope of assembly a. Figure 3 depicts a simple assembly to illustrate key points in the following discussion.

A connection, or connector, is established between two components when some source pin $c_i.r$ is connected to some sink pin $c_j.s$, $i \neq j$ (this inequality is assumed in further discussion). In text, we denote the connection as $c_i.r \rightsquigarrow c_j.s$. Components c_i and c_j must be contained by the same assembly if they are to be composed. A connection $c_i.r \rightsquigarrow c_j.s$ also requires that $c_i.r$ and $c_j.s$ be mutually conformant. The rule for mutual conformance is simple: both pins must be synchronous, or both must be asynchronous, and their data interfaces must have the same argument types, modes, and positions. Graphically, we denote a connection as a double-headed lollipop, with the lollipop heads circumscribing the connected pins.

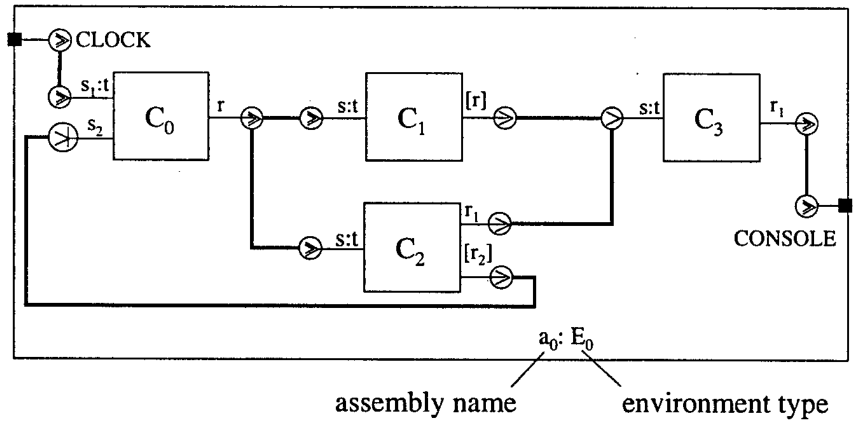


Figure 3: Graphical Notation for an Assembly

Each assembly has an associated environment type, which is denoted by the $:E_j$ suffix of an assembly name. Environment types play two crucial roles in Pin.

First, they define the services, specified as sink and source pins, that components may use. Graphically, these services are attached to the assembly by means of *environment junctions*, drawn as small black boxes. For example, in Figure 3, a runtime environment associated with assembly a_0 provides two services: the source pin $CLOCK$ and the sink pin $CONSOLE$. These services can be thought of as being implemented by an environment-provided component with a single sink pin ($a_0.CONSOLE$) and source pin ($a_0.CLOCK$).

We denote a connection with an environment service in an analogous way to that defined earlier, that is, as $a_0.CLOCK \leadsto a_0.c_0.s_1$, and $a_0.c_3.r_1 \leadsto a_0.CONSOLE$. To make these expressions easier to read, we allow assembly names to distribute over $\leadsto$, so, instead of the above, we could write $a_0(CLOCK \leadsto c_0.s_1)$ and $a_0(c_3.r_1 \leadsto CONSOLE)$. If the scope of a_0 is understood, we often omit it altogether.

Second, environment types supply the interaction models implemented by connectors. For example, consider the interaction topology in Figure 3, which includes

- $a_0(CLOCK \leadsto c_0.s_1)$, $a_0(c_3.r_1 \leadsto CONSOLE)$, and $a_0(c_2.r_2 \leadsto c_0.s_2)$
- $c_0.r \leadsto \{c_1.s, c_2.s\}$
- $\{c_1.r, c_2.r_1\} \leadsto c_3.s$

where we use sets $\{c.p_1, c.p_2, ...\}$ to denote multiple sinks and sources on $1\!:\!N$ and $N\!:\!1$ compositions, respectively. The semantics of the $1\!:\!1$ composition in the first bullet is likely to be intuitively clear, but what about the $1\!:\!N$ asynchronous composition in the second bullet? Does $c_0.r \leadsto \{c_1.s, c_2.s\}$ specify one or two connectors? Does this represent a broadcast or a sequence of unicasts? If it's a sequence, what is the order? What is the semantics of message buffering—FIFO or LIFO? What is the capacity of the message buffers? Analogously, what is the interaction order of the $N\!:\!1$ synchronous composition in the third bullet? We may want different answers to these questions in different component runtimes, and these different answers will result in different connector semantics. So, connectors are always defined with respect to some environment type.

3.7 Behavior: Reaction and Interaction

So far, we have focused on the syntax of composition. This makes sense, since we have largely been discussing constructive assemblies that are concerned with enabling runtime interactions rather than their behavior. Informally, Pin gives us the vocabulary to describe the structure of an assembly (i.e., the topology of the components and pins), while the compositional reasoning associated with analysis views describe what the assembly does at runtime. We say that CL is semantically extensible since different syntactic elements of CL—the language for documenting architectures in the Pin style—may be annotated with information that is used to construct, via interpretations, additional semantics for each assembly in the Pin style.

However, one behavioral semantics is fundamental to the meaning of many interpretations and so is incorporated into the core CL. Component and interaction behavior specifiable in a process algebra such as CSP [Hoare 85], CCS [Milner 89], π-Calculus [Milner 99], or FSP [Magee 01] must be common to every interpretation. We have elected to use CSP as the behavior specification language for CL. In this section, we describe only the essence of component behavior and its composition; the details are provided in Section 4. We treat the specification of component behavior first and then turn to composition behavior.

3.7.1 Component Behavior: Reactions

Components have intrinsic behavior: they are, after all, *implementations.* We model the behavior of a component in terms of *reactions.* A reaction is a CSP process that relates one or more sink pins to one or more source pins, indicating how the component reacts to stimulation of its sink pins. The general form is a process that looks something like $R = s \rightarrow r \rightarrow R$, where s is a sink pin that can be stimulated, r is a source pin that is stimulated in response to a stimulus on s, and R is the CSP process that describes this pattern of behavior. Note that because reactions are defined within the scope of a component, we omit usual scoping notation such as $c.s$, $c.r$, and $c.R$.

Reactions reflect the thread structure of a component. For example, all behavior implemented by a common thread is modeled as a single reaction. This allows analyses to take into account the actual degree of threading and potential concurrency errors of the component implementation. The behavior of a component as a whole is specified as the CSP parallel $\|$ or interleaved $|||$ composition of its reactions, depending on which better models the actual interaction among the component's threads. The gist of reaction rules is depicted in Figure 4.

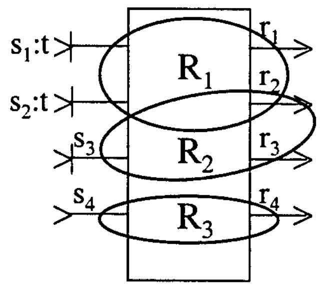


Figure 4: Reactions Partitioning the Behavior of a Component

In this example, there are three reactions: R_1, R_2, and R_3. The ovals are used to illustrate which pins are related by each reaction; for example, R_1 is shown as relating sink pins s_1 and s_2 to source pins r_1 and r_2. R_1 represents the behavior of a single thread t that is shared by sinks s_1 and s_2. A definition of R_1 could be $R_1 = (s_1 \rightarrow r_1 \rightarrow R_1) \square (s_2 \rightarrow r_2 \rightarrow R_1)$.

Reactions allow us to specify the causal dependencies among behaviors in an assembly of components. The most basic causal dependency is the dependency chain, as illustrated in the above reaction. More complex behaviors, such as coordination among reactions or changes in behavior based on accumulated state information, can also be modeled.

3.7.2 Assembly Behavior: Interactions

Up to this point, we have been using the $\rightsquigarrow$ operator in an informal manner. Although we never made the claim, a reasonable inference on the part of the reader would be that $\rightsquigarrow$ has some sort of composition semantics. We need more detail to justify this inference and outline its implications for understanding the behavior of assemblies, or, more technically, how component behaviors interact when composed using $\leadsto$.

A simple algebraic model of composition might be $\leadsto$: $C \times C \rightarrow C$, where we take C to denote the set of all components. That is, C is the carrier for an algebra with a single $\leadsto$ operator. We denote this simple algebra as $\langle C, \leadsto \rangle$. To give a concrete example, the pipeline style satisfies this model, where in place of $\langle C, \leadsto \rangle$, we have $\langle P, | \rangle >$, where P denotes Unix processes and | denotes the pipeline connector [Garlan 93].

However, Pin is more general and complex than the simple Unix pipeline model. For example, the constructive meaning of | in $P_1 \mid P_2$ is that the output (stdout) of P_1 is connected to the input (stdin) of P_2. Since | always and only connects *stdout* and *stdin*, there is no need to explicitly name these channels. In Pin, however, a component can have many input and output channels, so we need to make them explicit.

A generalization of the pipeline style to multiple input and output channels can be found in Pin and its predecessors, WaterBeans [Plakosh 99] and ComTek [Hissam 02]. The equivalent to a pipeline in Pin, at least syntactically, would be $P_1.stdout \leadsto P_2.stdin$. Note, however, that $P_1.stdout$ and $P_2.stdin$ denote a source and sink pin, respectively, not components. This is not just a lexicographical distinction, but reflects two critical Pin generalizations of the pipeline style:

- Components are no longer primitive; they are a composite of reactions, where we use the CSP algebraic operators || and ||| to specify composite behavior.
- Composition is generalized from data flow to arbitrary interaction schemes, such as, (a-)synchronous, connection(less), (uni/multi-)cast, and (2/N-)party rendezvous.

As a result, in place of the simple algebraic model $\langle C, \leadsto \rangle$ we have $\langle R, \overset{a^n}{\leadsto} | \; a^n \in E \rangle$, where, in place of components, we have reactions R, and in place of the single operator $\leadsto$, we have a set of operators $\overset{a^n}{\leadsto}$, one for each interaction scheme a defined for an environment E, where the operators may have arbitrary arity (i.e., interactions may involve an arbitrary number of parties).

So, for example, from Figure 3, the N:1 interaction $a_0(c_0.r \overset{\gg}{\leadsto} \{c_1.s, c_2.s\})$ denotes the syntactic composition of three components c_0, c_1, and c_2, in assembly a_0, on pins $c_0.r$, $c_1.s$ and $c_2.s$. The interaction scheme for this composition is $\gg$, as defined in environment type E_0. We can not tell whether $\overset{\gg}{\leadsto}$ is a 2-ary (unicast) or N-ary (broadcast) operator. This and other aspects of the semantics of this interaction must be formally specified.

3.8 Hierarchical Assembly

So far, we have described a component model that is quite flat: components can interact only with components in the same assembly and only within a single runtime environment associated with that assembly. Pin will not scale to interesting problems without introducing some form of hierarchical composition. In fact, the algebraic model has a hierarchical meaning, as $R_3 = R_1 \leadsto R_2$, for some arbitrary binary composition operator $\leadsto$ and reactions $R_{\{1,2,3\}}$, induces a hierarchy that is rooted at R_3 and contains R_1 and R_2 as leaves.

Hierarchical assembly in Pin always involves treating an assembly as a component. That is, the assembly has an interface defined in terms of source and sink pins. The correspondence between component pins and assembly pins is established by means of assembly junctions. Two forms of junctions are currently defined: *null* junctions and *gateway* junctions. We note at the outset that hierarchical composition introduces many subtle complexities not generally addressed by component technology. Many of these subtleties, by intent, lie beneath the surface of the following discussion.

3.8.1 Null Junctions

A null junction has no behavior, hence the name. A null junction is an abstraction mechanism that is related to the hiding operator in process algebras such as CSP and CCS. That is, all the pins that do not appear as connected to null junctions are hidden from the external world. This is the only form of hierarchical composition that is supported by composition environments such as FSP [Magee 01], Koala [van Ommering 02], and Tracta [Giannakopoulou 99] (all of these are based on the original work on Darwin [Magee 93]). The use of null junctions is illustrated in Figure 5. In this figure, we treat a_0 as a subassembly with an interface consisting of a single sink pin $a_0.s$ and a single source pin $a_0.r$. In this example, $a_0.r$ is an alias for $a_0.c_0.r$, and the real connector is $a_2(a_0.c_0.r \leadsto a_1.c_0.s_1)$. Using null junctions, we can treat a_0 and a_1 as components and write the interaction as $a_2(a_0.r \leadsto a_1.s_1)$.

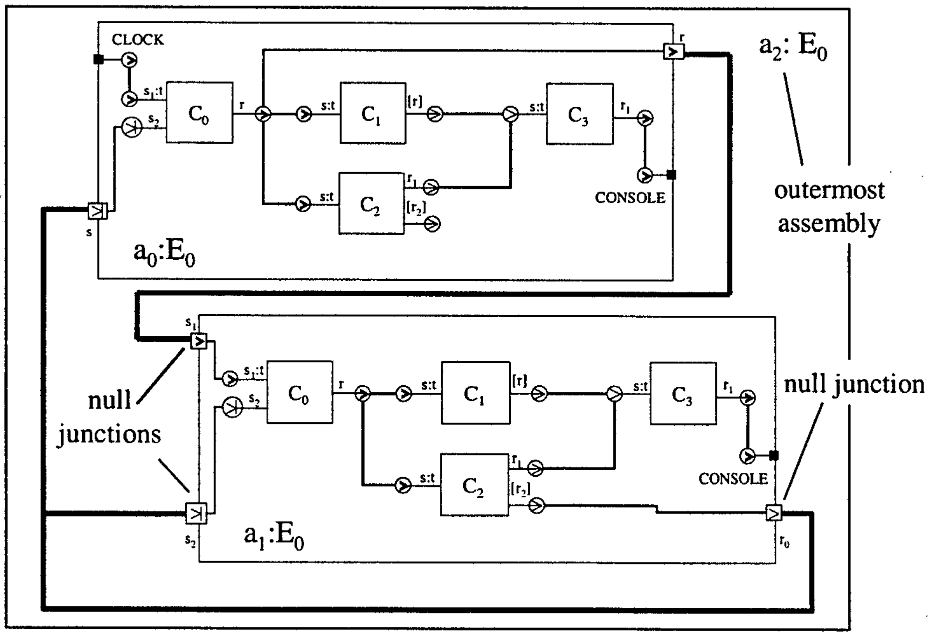


Figure 5: Intra-Assembly Hierarchy with Null Junctions

Hierarchical assembly using null junctions is useful strictly for analysis purposes when there is a defined equivalence relation that permits one assembly to be replaced by another that is behaviorally equivalent with respect to some property. For example, various trace-based equivalences are defined for CSP and denoted as refinement relations; in CCS, there are various observational equivalences, denoted as simulation relations. Variations of refinement and simulation preserve different kinds of properties. Given an equivalence relation that preserves the property of, say, deadlock freedom, we can be sure that a demonstrably deadlock free assembly can be replaced with an assembly that is behaviorally equivalent with respect to deadlock, but whose behavioral model is otherwise more abstract.

Note that the assemblies in Figure 5 are all of environment type E_0. The stipulation that null junctions have no behavior means that they do not introduce interactions; they introduce only aliases. This means that the null connectors used in $a_2(a_0.r \rightsquigarrow a_1.s_1)$ can be used only to connect assemblies having the same environment type, in the context of an enclosing assembly of that same environment type. This restriction is relaxed by gateway junctions.

3.8.2 Gateway Junctions

Assemblies in different types of environments are composed using gateway junctions. Unlike null junctions, gateway junctions do have behavior. Their purpose is to provide a bridge from the interaction schemes (and connectors) defined in one environment type to those defined in some other environment type. The main idea is illustrated in Figure 6, which recasts the previous illustration (Figure 5) so that assemblies a_0 and a_1 have different environment types, E_0 and E_1, respectively and are composed in an assembly having a still different environment type, E_2. Strictly speaking, it is possible for $E_0 = E_1$; however, for gateways to make sense, $E_0 \neq E_2$ and $E_1 \neq E_2$.

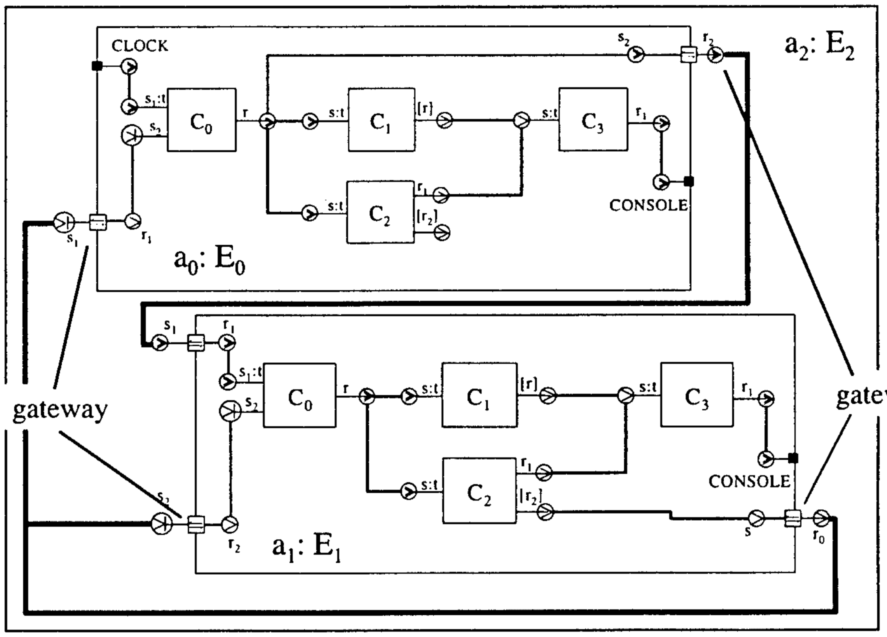


Figure 6: *Intra-Assembly/Inter-Assembly Hierarchy with Gateways*

In the example in Figure 6, the gateway $a_0.r_2$ translates $\overset{\gg}{\leadsto}$ interactions, as they are defined in environment type E_0, to $\overset{\gg}{\leadsto}$ interactions, as they are defined in environment type E_2. The interaction $a_2(a_0.r_2 \leadsto a_1.s_1)$ is enabled using an operator (connector) $\overset{\gg}{\leadsto}$ defined in environment type E_2.

From the composition perspective, the interactions between a_0 and a_1 are semantically indistinguishable from those that would arise if either or both a_0 and a_1 were components native to E_2 and not assemblies that had been transplanted, via gateways, into E_2.

The notion of composition across heterogeneous environment types is quite similar to what Szyperski described as tiered frameworks [Szyperski 97]. Terminology and other subtle distinctions aside, it is surprising that there are, as far as the authors know, no commercial or research component technologies (other than Pin) that realize this idea.

4 Formal Model of CL

The formal model for the behavioral semantics of CL is based on the CSP process algebra [Hoare 85, Roscoe 98]. CSP is used to describe the behavior of components, as well as the interactions among them.

4.1 Conceptual Model

One goal of compositional reasoning is to combine the behavior of interacting components to produce a single description that explains the behavior of the composition. Figure 7 shows a simple way that composition could be approached in CL. The behaviors of components c_1 and c_2 are specified as CSP processes P_{c_1} and P_{c_2}, respectively. Pins show up in these processes as CSP events, and two processes interact whenever they synchronize on an event. The simplest form of composition would be to put these processes in parallel over the events representing pins, defining the composition as $P_{c_1} \parallel P_{c_2}$.

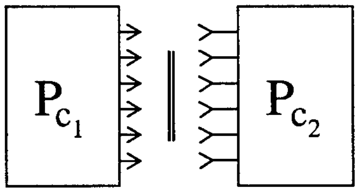


Figure 7: A Simple Approach to Composition Semantics

This approach is too simple though. It represents all interactions as synchronizations of source pin events with sink pin events, that is, synchronous communication. In practice, components communicate using several different types of communication, and the differences matter.

Figure 8 illustrates a more flexible way to model composition. c_1 and c_2 have CSP behavior specifications as above, but rather than composing them directly, an additional CSP process is interposed to represent the semantics of different types of interactions.

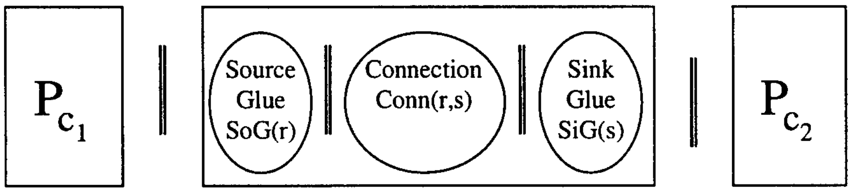


Figure 8: Use of Glue to Model Interaction Behavior

The CSP process representing interaction semantics is structured as the composition of three processes: a source glue, a sink glue, and a connection between them. The source glue describes the interaction semantics from the source's perspective (e.g., describing that a synchronous source expects an acknowledgment as part of each interaction). The sink glue describes the interaction semantics from the sink's perspective (e.g., describing that asynchronous interaction requests are queued until an asynchronous sink pin is ready for the next request). The connection binds a particular source glue to a particular sink glue, reflecting the topology of the interaction.

This approach, while an improvement, is incomplete since we have not yet addressed implementation concurrency. Is concurrent activity possible within a component? Can multiple components concurrently interact with the same sink pin of another component?

We could try to avoid these issues by dictating that all sink pins of a component be threaded and share the same thread of execution. But this restriction is not very satisfactory: it means that we cannot model reentrant sink pins, which explicitly permit multiple concurrent interactions. It also means a loss of useful analysis results. Errors in concurrent systems are often the result of race conditions among concurrent tasks, conditions that are notoriously difficult to detect in testing. Model checking, on the other hand, is well suited to detect such errors; but, we cannot detect them if we do not model the inherent concurrency in the first place.

Therefore, we address implementation concurrency explicitly in our semantic model. We use two concepts–reactions and logical threads–to structure a component's behavior specification in a way that models the concurrent threads of execution found in its implementation.

Each reaction has a specific interpretation in terms of the threadedness of its component. All sink pins handled by the same thread are collected into a single reaction that defines the behavior of that thread. Each sink pin that is not threaded is specified by its own reaction, which defines the behavior that is executed in the caller's thread. These interpretations lead to the following constraints on reactions:

- Every sink pin must appear in exactly one reaction.
- A source pin may appear in more than one reaction, but every source pin must appear in at least one reaction.
- All sink pins appearing in the same reaction must be handled by the same thread.
- All sink pins handled by the same thread must appear in the same reaction.

However, the interpretation of reactions does not fully address reentrant sink pins. Can multiple interactions occur concurrently on a reentrant sink pin? The answer is, of course, yes, but modeling the reaction for a reentrant sink pin as a single CSP process does not accurately model this concurrency.

The simplest solution is to copy the reaction N times to permit up to N concurrent interactions involving a reentrant sink pin. We call each such copy of a reaction a *logical thread*. For reentrant sink pins, many logical threads may correspond to the same reaction. For all other reactions, exactly one logical thread corresponds to each reaction.

A logical thread is a CSP process modeling a thread of execution that is implemented by the component, regardless of whether that thread of execution occurs on a thread managed by that component. When describing interactions among components, we use a definition of a component's behavior that is structured in terms of logical threads, rather than reactions. This is a simple transformation of the user's specification, which is structured in terms of reactions and models the true concurrency of the component.

Alternatively, we could require that component specifications already have the correct number of copies of reentrant sink pin reactions. But, what is the correct number of copies to make? Unfortunately, the answer is: it depends. The number of copies should be the same as the number of logical threads of other components that can interact with the reentrant sink pin, and that number depends on what is in the component's environment. Without knowing a component's environment, a component specifier cannot determine the number of copies to make, and if the component is to be used in several contexts, the number will vary with context. As such, relating reactions to logical threads is a task best performed when an environment has been specified (i.e., when we know what other components interact with the component), rather than when a component is specified.

Returning to the example illustrated in Figure 4, R_3 is a reaction for a reentrant sink pin s_4. When the component interacts with only one other component, and that component has two logical threads that interact with s_4, there are two logical threads for pin s_4, each of which is a copy of R_3.

4.2 Formal Definitions

4.2.1 Definition of Sink Pin

A sink pin is represented as a CSP event of the same name. Use of a sink pin is represented by a pair of these events, one denoting the initiation of an interaction on that pin (e.g., a function being called) and the other denoting the completion of an interaction on that pin (e.g., a function returning). Any other interactions or activities that are internal to a component and that are performed before completing the interaction will appear as events between those two events. Non-sink events may appear after the second occurrence of the sink event only if the sink pin has a thread.

A reaction describing the behavior of an interaction on reentrant sink pin s that engages in an interaction on source r before completing the interaction on sink s would be written as: $P = s \rightarrow r \rightarrow r \rightarrow s \rightarrow P$. It would be illegal to put any event after the second s, since a reentrant sink pin never has a thread.

A sink pin may represent an interaction involving in/out data parameters. This is represented using CSP compound events, where the sink pin name is the name of the channel, and data parameters are data components appended to the channel using normal CSP conventions (i.e., $e!x$ represents event e that supplies data value x, and $e?y$ represents event e that accepts a data value y).

The first occurrence of a sink event in a reaction may include input data parameters (e.g., $s?x?y$), but may not include output data parameters. The second occurrence of a synchronous sink event in a reaction may include output data parameters (e.g, $s!x!y$), but may not include input data parameters. The second occurrence of an asynchronous sink event may not include any data parameters.

4.2.2 Definition of Source Pin

A source pin is also represented as a CSP event of the same name. Use of a source pin is represented by a pair of these events, one denoting the initiation of an interaction on that pin (e.g., calling a function) and the other denoting the completion of an interaction on that pin (e.g., the return of the function). No other events may appear between these two events.

A reaction describing the behavior of an interaction on sink pin s that engages in an interaction on source r before completing the interaction on sink s would be written as: $P = s \rightarrow r \rightarrow r \rightarrow s \rightarrow P$. It would be illegal to put any event between the two occurrences of r in the reaction.

As with sink pins, source pins may represent an interaction involving in/out data parameters, and the same CSP representation is used. The first occurrence of a source event in a reaction may include output data parameters (e.g., $r!x!y$), but may not include input data parameters. The second occurrence of a synchronous source event in a reaction may include input data parameters (e.g., $r?x?y$), but may not include any output data parameters. The second occurrence of an asynchronous source event may not include data parameters.

4.2.3 Definition of Reaction

A reaction is represented as a CSP process. The initial set of events accepted by that process should be the sink pins described by the reaction. Reactions are transformed into logical threads when a component is composed with other components (i.e., when the environment of the component is known).

4.2.4 Definition of Logical Thread

A logical thread, which is represented as a CSP process, is a copy of a reaction that executes in a particular context. A reaction for a component's thread will equal a logical thread. A reaction for a reentrant sink pin may be transformed into multiple logical threads, one per context that can interact with the pin.

4.2.5 Definition of Component

A component is represented by the 3-tuple $\langle \mathcal{S}, \mathcal{R}, \mathcal{P} \rangle$ defined below.

- $\mathcal{S}$ is the set of the component's sinks.
- $\mathcal{R}$ is the set of the component's sources.
- $\mathcal{P}$ is the component's behavior, specified as a CSP process. $\mathcal{P}$ is structured as the composition (using $\|$ or $|||$) of the component's reactions together with any auxiliary CSP processes needed to coordinate the reactions (e.g.,

to model a lock used by multiple reactions). An auxiliary CSP process is defined to be any process that does not refer to any of the component's pins.

4.2.6 Definition of Assembly

An assembly is represented by the 4-tuple $\langle \mathcal{AC},\ top,\ \mathcal{T},\ threadmap \rangle$ defined below.

- $\mathcal{AC}$ is the finite set of components in the assembly. No two components within an assembly can have the same name.
- *top* is the finite set of $\langle$*source pin*, *sink pin*$\rangle$ pairs defining the assembly's topology. Each pair represents a source pin that interacts with a sink pin. The value of *top* is supplied by the component assembler, but must conform to the following constraints:

  $\forall (c_i.r, c_j.s) \in top \bullet c_i \neq c_j \wedge r \in c_i.\mathcal{R} \wedge s \in c_j.\mathcal{S}$

  That is, every connection must be between a source pin and a sink pin of two different components.

  $\forall\, c_i.r, c_j.s_i \bullet ((c_i.r, c_j.s_i) \in top \wedge Synchronous(c_i.r)) \Rightarrow$
  $(\neg \exists\, c_k, s_j \bullet (c_i.r, c_k.s_j) \in top \wedge c_k.s_j \neq c_j.s_i))$

  That is, if a source pin is synchronous, it cannot be connected to more than one sink pin.

- $\mathcal{T}$ is the finite set of $\langle$*pin*, *logical thread*$\rangle$ pairs defining the concurrency of pins in the assembly. Each pair represents a logical thread of the pin's component that can interact via the pin. The value of $\mathcal{T}$ is calculated based on components' reactions and an understanding of the threadedness of the components' environments, and satisfies the following constraints:

  $\forall (c.p, c.t) \in \mathcal{T} \bullet p \in (c.\mathcal{S} \cup c.\mathcal{R}) \wedge t$ is a logical thread of c

  That is, each logical thread of a component interacts only via pins of that component.

  $\forall\, c.p \bullet (p \in c.\mathcal{S} \wedge p$ is not reentrant$) \Rightarrow (\#\{t \mid (c.p, c.t) \in \mathcal{T}\} = 1)$

  That is, all sink pins that are not reentrant have exactly one logical thread.

- *threadmap* is a finite set of $\langle$*logical thread*, *logical thread*$\rangle$ pairs. Each pair represents a logical thread of one component that interacts (via source and sink pins) with a logical thread of another component. Like $\mathcal{T}$, the value of *threadmap* is calculated based on components' reactions and an understanding of the threadedness of the components' environments. The calculated value of *threadmap* must satisfy the following constraints:

  $\forall (c_i.t_i, c_j.t_j) \in threadmap \bullet \exists\, r, s \bullet (c_i.r, c_j.s) \in top\ \wedge (c_i.r, c_i.t_i) \in \mathcal{T}\ \wedge$
  $(c_j.s, c_j.t_j) \in \mathcal{T}$

  That is, every mapping between logical threads must be between logical threads of two different components and must correspond to the logical threads of a pair of pins connected in *top*.

$$\forall\, c_i.r, c_j.s, c_i.t_i \bullet (r \in c_i.\mathcal{R} \wedge (c_i.r, c_i.t_i) \in \mathcal{T} \wedge (c_i.r, c_j.s) \in \mathit{top}) \Rightarrow$$
$$(\exists\, c_j.t_j \bullet (c_i.t_i, c_j.t_j) \in \mathit{threadmap} \wedge (c_j.s, c_j.t_j) \in \mathcal{T})$$

That is, every logical thread for a source pin that interacts with some sink pin must be mapped to at least one logical thread of that sink pin.

$$\forall\, c_i.r, c_j.s, c_j.t_j \bullet (s \in c_j.\mathcal{S} \wedge (c_j.s, c_j.t_j) \in \mathcal{T} \wedge (c_i.r, c_j.s) \in \mathit{top}) \Rightarrow$$
$$(\exists\, c_i.t_i \bullet (c_i.t_i, c_j.t_j) \in \mathit{threadmap} \wedge (c_i.r, c_i.t_i) \in \mathcal{T})$$

That is, every logical thread for a sink pin that interacts with some source pin must be mapped to at least one logical thread of that source pin.

$$\forall\, c_i.r, c_j.s, c_j.t_j \bullet (s \in c_j.\mathcal{S} \wedge (c_j.s, c_j.t_j) \in \mathcal{T} \wedge c_j.s \text{ is a reentrant sink pin} \wedge$$
$$(c_i.r, c_j.s) \in \mathit{top}) \Rightarrow (\exists\, c_i.t_i \bullet (c_i.t_i, c_j.t_j) \in \mathit{threadmap} \wedge (c_i.r, c_i.t_i) \in \mathcal{T} \wedge$$
$$\forall\, c_k.t_k \bullet (c_k.t_k, c_j.t_j) \in \mathit{threadmap} \Rightarrow (c_k.t_k = c_i.t_i))$$

That is, every logical thread for a reentrant sink pin that interacts with some source pin must be mapped to exactly one logical thread.

4.2.7 Definition of Interaction

An interaction is defined in the context of a particular assembly and is represented by a $\langle$*source pin*, *sink pin*$\rangle$ pair that is an element of *top*. Source r of component c_i is the source pin participating in the interaction. Sink s of component c_j is the sink pin participating in the interaction.

4.3 Composition Semantics

When composing two components, c_1 and c_2, we define the semantics using the CSP parallel composition of their behavior specifications, P_{c_1} and P_{c_2}. However, as mentioned earlier, it's not as simple as defining the composition to be $P_{c_1} \parallel P_{c_2}$. There are other important issues to consider.

First, we need to transform P_{c_1} and P_{c_2} from compositions of reactions to compositions of logical threads. Then, we need to carefully plan the synchronization between CSP processes such that synchronization occurs only on certain events. To accomplish this, we apply an event-renaming convention that gives different names to events that should not be synchronized. Finally, we need to interpose CSP processes that model the behavior of the different types of interactions that can occur between components.

The following subsections address these topics. We start by addressing the necessary transformations to component behavior descriptions (i.e., reaction to logical thread transformation and event renaming). This is followed by formally defining the different types of interactions between a source pin and a sink pin. Finally, we present the general model for defining the behavior of an assembly of components with an arbitrary finite set of interactions.

4.3.1 Composition Semantics: Components

As mentioned earlier, a component is described by a CSP behavior specification P_c, which is structured in terms of CSP processes modeling its reactions. This behavior specification needs to be transformed in two ways to exhibit the correct semantics when combined in an interaction or assembly.

First, we need to transform reactions into logical threads. For reactions that do not model reentrant sink pins, the logical thread is equal to the reaction. For reactions that model a reentrant sink pin, we need to replace the reaction with N copies of the reaction, each of which is a distinct logical thread and models the possibility of a different concurrent interaction with the sink pin. For the reaction for reentrant sink pin $c.s$, $N = \#\{t \mid (c.s, c.t) \in \mathcal{T}\}$.

For example, consider the composition illustrated in Figure 9. The reaction of reentrant sink pin s of component c_1 is given by $R_s = s \rightarrow x \rightarrow x \rightarrow s \rightarrow R_s$. The behavior specification of the sink pin's component is defined as $P_{c_1} = R_s \mid\mid\mid R_t$, where R_t is another reaction of the component that handles non-reentrant sink pin t. In this assembly, c_1 is composed with c_2. c_2 has two logical threads that use a source pin connected to c_1's reentrant sink pin; consequently, we need two copies of the reaction that handles activity on reentrant sink pin s (i.e., two logical threads). The transformed behavior of c_1 could be rewritten as $P'_{c_1} = (\mid\mid\mid_{t \in \{t_1, t_2\}}\ t : R_s) \mid\mid\mid R_t$. This process has two copies of R_s, each with a different prefix corresponding to a different logical thread.[3]

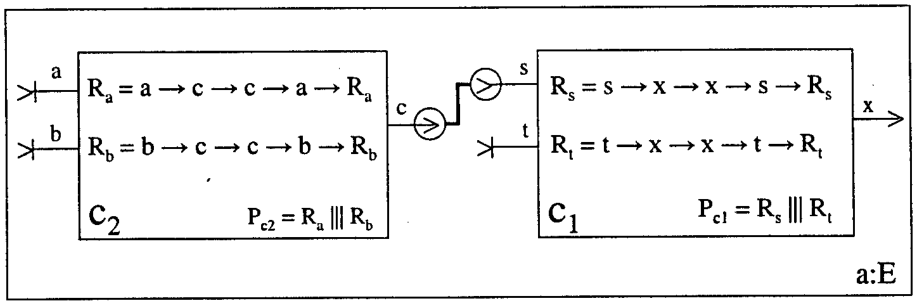

a
b
$R_a = a \rightarrow c \rightarrow c \rightarrow a \rightarrow R_a$
$R_b = b \rightarrow c \rightarrow c \rightarrow b \rightarrow R_b$
C_2
$P_{c2} = R_a \mid\mid\mid R_b$
c
s
t
$R_s = s \rightarrow x \rightarrow x \rightarrow s \rightarrow R_s$
$R_t = t \rightarrow x \rightarrow x \rightarrow t \rightarrow R_t$
C_1
$P_{c1} = R_s \mid\mid\mid R_t$
x
a:E

Figure 9: Composition of Two Components

Second, we need to apply an event-renaming convention to P_{c_1} to ensure proper synchronization. We use a convention that gives each pin used by each component's logical thread a unique name. For example, a source pin that is used by two different logical threads is renamed by prefixing the name of the source pin with the name of the component and logical thread in which it used. Thus, each logical thread has its own copy of the event (e.g., $c_1.t_1.s$ and $c_1.t_2.s$).

Event renaming is important when c is composed with another component. Interactions on a

[3] This is a simplification. By relabeling all events in R_s with a prefix unique to the logical thread, problems could be introduced if R_s contains any events that are internal to the component. For example, if several logical threads use an internal event to coordinate access to a shared resource within the component, relabeling this event differently in each logical thread would incorrectly eliminate synchronization among the logical threads.

source pin that occur in one logical thread must be distinguished from interactions on the same source pin occurring in other logical threads. In particular, if the source pin uses input and output data parameters, we must distinguish between two interactions to correctly associate related input and output data parameters.

For a given logical thread, $c.t$, which is a transformation of reaction R, the behavior of logical thread $c.t$ is defined by the process LT_t, where

$$LT_t = R\,[\forall\, p \in (c.\mathcal{S} \cup c.\mathcal{R}) \bullet c.t.p\backslash p, \forall\, e \in (\alpha R \setminus (c.\mathcal{S} \cup c.\mathcal{R})) \bullet c.p\backslash p]$$

Formally, we define the fully transformed behavior specification of component c as P'_c, where P'_c is produced by making substitutions in P_c. As mentioned earlier, P_c is required to be structured as a composition of CSP processes, including both reactions and auxiliary CSP processes. In deriving P'_c, we retain the original composition semantics (the choice of combining CSP processes using $\|$ or $|||$) among the CSP processes, but we transform each reaction into its corresponding logical thread(s). To do this, we iterate through each CSP process P in the composed process P_c and make the following substitution:

- If the process represents a reaction, R, that does not describe the behavior of a reentrant sink pin, substitute LT_t for R.
- If the process represents a reaction, R, that describes the behavior of reentrant sink pin $c.s$, substitute $|||_{t|(c.s,c.t)\in\mathcal{T}}\ LT_t$ for R.
- If the process represents an auxiliary process, substitute P for P (i.e., make no change).

4.3.2 Composition Semantics: Interactions

Interaction $c_i.r \rightsquigarrow c_j.s$ is a connection between source r of component c_i and sink s of component c_j. Formally, we represent such an interaction as

$$I(c_i.r, c_j.s) = P'(c_i) \parallel Glue(c_i.r, c_j.s) \parallel P'(c_j)$$

where $Glue(c_i.r, c_j.s)$ is a process that describes the semantics of the interaction shared between $c_i.r$ and $c_j.s$, and is defined as

$$\begin{aligned} Glue(c_i.r, c_j.s) = {} & SoG(c_i.r) \parallel \\ & (|||_{t_i,t_j|(c_i.r,c_i.t_i)\in\mathcal{T}\wedge(c_j.s,c_j.t_j)\in\mathcal{T}\wedge(c_i.t_i,c_j.t_j)\in \mathit{threadmap}}\ Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)) \\ & \parallel SiG(c_j.s) \end{aligned}$$

where $SoG(c_i.r)$, the source glue for source r of component c_i, is the process that defines the interaction semantics from the source's perspective (e.g., describing that a synchronous source expects an acknowledgment as part of each interaction). It takes into account that there might be multiple logical threads in c_i on which interactions on r might occur concurrently.

$SiG(c_j.s)$, the sink glue for sink s of component c_j, is the process that defines the interaction semantics from the sink's perspective (e.g., describing that asynchronous interaction requests are queued until an asynchronous sink pin is ready for the next request). It takes into account that there might be multiple logical threads in c_j on which interactions on s might occur concurrently.

$Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)$ is the process that binds one logical thread $c_i.t_i$ of a source r of component c_i to one logical thread $c_j.t_j$ of a sink s of component c_j. This binding is accomplished by matching *left* events shared with a particular source glue to *right* events shared with a particular sink glue.

The semantics of an interaction vary with the type of interaction, so $SoG(c_i.r)$, $SiG(c_j.s)$, and $Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)$ are defined differently depending on whether each interaction is reentrant, mutex, or asynchronous.

Reentrant Interactions

The process definitions for reentrant interactions are as follows:

$$\begin{aligned}
&SoG(c_i.r) = |||_{t_i|(c_i.r,c_i.t_i)\in\mathcal{T}}\ SoG(c_i.r, c_i.t_i)\\
&SoG(c_i.r, c_i.t_i) = c_i.t_i.r \rightarrow c_i.t_i.r.left \rightarrow c_i.t_i.r.left \rightarrow c_i.t_i.r \rightarrow SoG(c_i.r, c_i.t_i)\\
&Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j) = c_i.t_i.r.left \rightarrow c_j.t_j.s.right \rightarrow c_j.t_j.s.right \rightarrow c_i.t_i.r.left \rightarrow\\
&\qquad\qquad Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)\\
&SiG(c_j.s) = |||_{t_j|(c_j.s,c_j.t_j)\in\mathcal{T}}\ SiG(c_j.s, c_j.t_j)\\
&SiG(c_j.s, c_j.t_j) = c_j.t_j.s.right \rightarrow c_j.t_j.s \rightarrow c_j.t_j.s \rightarrow c_j.t_j.s.right \rightarrow SiG(c_j.s, c_j.t_j)
\end{aligned}$$

For reentrant interactions, the mapping between logical threads of a source pin and logical threads of a sink pin is one to one. As such, we could uniquely identify *right* events to the sink glue either by prefixing them with source pin information (component, logical thread, and pin names) or by prefixing them with sink pin information. We opt for sink pin information as it allows a simpler $SiG(c_j.s)$ process definition.

Mutex Interactions

The process definition of $SoG(c_i.r)$ for mutex interactions is the same as for the reentrant case. The remaining process definitions for mutex interactions are as follows:

$$\begin{aligned}
&Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j) = c_i.t_i.r.left \rightarrow c_i.t_i.r.right \rightarrow c_i.t_i.r.right \rightarrow c_i.t_i.r.left \rightarrow\\
&\qquad\qquad Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)\\
&SiG(c_j.s) = SiG(c_j.s, c_j.t_j),\text{ where } t_j = \text{the only element of } \{t \mid (c_j.s, c_j.t) \in \mathcal{T}\}\\
&SiG(c_j.s, c_j.t_j) = WaitRight(c_j.s, c_j.t_j)_{<>}\\
&\quad WaitRight(c_j.s, c_j.t_j)_{<>} = \Box_{c_i.r|(c_i.r,c_j.s)\in top}\ c_i?t_i!r!right \rightarrow c_j.t_j.s \rightarrow\\
&\qquad\qquad WaitRight(c_j.s, c_j.t_j)_{<c_i.t_i.r>}\\
&\quad WaitRight(c_j.s, c_j.t_j)_{Q^\frown<x.y.z>} = (\Box_{c_i.r|(c_i.r,c_j.s)\in top}\ c_i?t_i!r!right \rightarrow\\
&\qquad\qquad WaitRight(c_j.s, c_j.t_j)_{<c_i.t_i.r>^\frown Q^\frown<x.y.z>})\\
&\qquad\qquad \Box\ c_j.t_j.s \rightarrow x.y.z.right \rightarrow Next(c_j.s, c_j.t_j)_Q\\
&\quad Next(c_j.s, c_j.t_j)_{<>} = WaitRight(c_j.s, c_j.t_j)_{<>}\\
&\quad Next(c_j.s, c_j.t_j)_{Q^\frown<x.y.z>} = c_j.t_j.s \rightarrow WaitRight(c_j.s, c_j.t_j)_{Q^\frown<x.y.z>}
\end{aligned}$$

For mutex interactions, the mapping between logical threads of a source pin and logical threads of a sink pin is many to one since there is always exactly one sink logical thread. As such, to uniquely identify *right* events to the sink glue, we must prefix them with source pin information.

Note that we have modeled the queue of waiting interactors as a FIFO queue. Other queuing policies are possible and could be introduced as variants of $SiG(c_j.s)$.

Asynchronous Interactions

The process definitions for asynchronous interactions are a little more complicated, because it is possible to connect an asynchronous source pin to multiple asynchronous sink pins. Whenever a source is connected to multiple sinks, the semantics call for the source to interact with all the connected sinks, not just a subset of them. Further, the interaction represents a sequence of two-way interactions, not a single N-way interaction.

Below, we define two alternative meanings for such an interaction. The first meaning, which we call non-deterministic, leaves the order of two-way interactions unspecified. The order can be different each time the source pin is activated. The second meaning, which we call deterministic, allows the order to be explicitly specified and so is the same each time the source pin is activated.

Non-Deterministic Semantics. The non-deterministic semantics are presented below:

$$\begin{aligned}
&SoG(c_i.r) = |||_{t_i|(c_i.r,c_i.t_i)\in\mathcal{T}}\ SoG(c_i.r, c_i.t_i)\\
&SoG(c_i.r, c_i.t_i) = c_i.t_i.r \rightarrow SendAll(c_i.r, c_i.t_i)_{conns}\ ;\ c_i.t_i.r \rightarrow SoG(c_i.r, c_i.t_i)\\
&SendAll(c_i.r, c_i.t_i)_{\emptyset} = Skip\\
&SendAll(c_i.r, c_i.t_i)_S = \sqcap_{(c_j.s,c_j.t_j)\in S}\ c_i.t_i.r.c_j.t_j.s.left \rightarrow SendAll(c_i.r, c_i.t_i)_{S\setminus\{(c_j.s,c_j.t_j)\}}\\
&conns = \{(c_j.s, c_j.t_j) \mid (c_i.r, c_j.s) \in top \wedge (c_i.r, c_i.t_i, c_j.s, c_j.t_j) \in threadmap \wedge\\
&\qquad (c_j.s, c_j.t_j) \in \mathcal{T}\}
\end{aligned}$$

$$Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j) = c_i.t_i.r.c_j.t_j.s.left \rightarrow c_j.t_j.s.right \rightarrow Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)$$

$$\begin{aligned}
&SiG(c_j.s) = SiG(c_j.s, c_j.t_j), \text{ where } t_j = \text{the only element of } \{t \mid (c_j.s, c_j.t) \in \mathcal{T}\}\\
&SiG(c_j.s, c_j.t_j) = WaitRight(c_j.s, c_j.t_j)_{<>}\\
&\quad WaitRight(c_j.s, c_j.t_j)_{<>} = c_j.t_j.s.right \rightarrow c_j.t_j.s \rightarrow WaitRight(c_j.s, c_j.t_j)_{<s>}\\
&\quad WaitRight(c_j.s, c_j.t_j)_{Q^\frown<x>} = c_j.t_j.s.right \rightarrow WaitRight(c_j.s, c_j.t_j)_{<s>^\frown Q^\frown<x>}\\
&\qquad\qquad\qquad\qquad\qquad\quad \square\ c_j.t_j.s \rightarrow Next(c_j.s, c_j.t_j)_Q\\
&\quad Next(c_j.s, c_j.t_j)_{<>} = WaitRight(c_j.s, c_j.t_j)_{<>}\\
&\quad Next(c_j.s, c_j.t_j)_{Q^\frown<x>} = c_j.t_j.s \rightarrow WaitRight(c_j.s, c_j.t_j)_{Q^\frown<x>}
\end{aligned}$$

Deterministic Semantics. The component assembler must specify additional information for this alternative. Specifically, the order in which the source will interact with each connected sink must be specified. The following CSP processes assume that this information is specified as a sequence of pairs $\langle c_j.s, c_j.t_j\rangle$ called *conns*. The definitions of $Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)$ and $SiG(c_j.s)$ are the same as for the non-deterministic alternative.

$$SoG(c_i.r) = |||_{t_i|(c_i.r,c_i.t_i)\in\mathcal{T}}\ SoG(c_i.r, c_i.t_i)$$

$$
\begin{aligned}
&SoG(c_i.r, c_i.t_i) = c_i.t_i.r \rightarrow SendAll(c_i.r, c_i.t_i)_{conns} \ ; \ c_i.t_i.r \rightarrow SoG(c_i.r, c_i.t_i)\\
&SendAll(c_i.r, c_i.t_i)_{<>} = Skip\\
&SendAll(c_i.r, c_i.t_i)_{Q^\frown<(c_j,t_j,s)>} = c_i.t_i.r.c_j.t_j.s.left \rightarrow SendAll(c_i.r, c_i.t_i)_Q
\end{aligned}
$$

In the context of an interaction between two asynchronous pins, the mapping between logical threads of a source pin and logical threads of a sink pin is many to one, since there is exactly one sink logical thread. However, in the context of an assembly, the mapping is many to many since a source pin can be connected to multiple sink pins. This means that to uniquely identify *right* events to the sink glue we must prefix them with source and sink information.

Again, we have modeled the queue of waiting interactors as a FIFO queue. Other queuing policies are possible and could be introduced as variants of $SiG(c_j.s)$. Careful inspection will reveal that the information being queued is not used beyond keeping tracking of how many requests are pending. This would not be the case if the interaction included input data parameters, however, as the parameters are queued as part of the s events. This enables the sink pin to handle different requests (i.e., those with different input parameters) in the correct order.

4.3.3 Composition Semantics: Assemblies

The composed behavior of an assembly of a finite set of components AC with connections as defined in *top*, logical thread allocation to pins as defined in $\mathcal{T}$, and logical thread connections as defined in *threadmap* is given by

$$
\begin{aligned}
&Assembly = \ Components(AC) \parallel SourceGlues(AC) \parallel SinkGlues(AC) \parallel Connections(AC)\\
&Components(AC) = |||_{c \in AC} \ P'_c\\
&SourceGlues(AC) = |||_{c.r \in AllSources} \ SoG(c.r)\\
&AllSources = \{c.r \mid c \in AC \wedge r \in c.\mathcal{R}\}\\
&SinkGlues(AC) = |||_{c.s \in AllSinks} \ SiG(c.s)\\
&AllSinks = \{c.s \mid c \in AC \wedge s \in c.\mathcal{S}\}\\
&Connections(AC) = |||_{(c_i.r, c_j.s) \in top} \ Conns(c_i.r, c_j.s)\\
&Conns(c_i.r, c_j.s) =\\
&\qquad |||_{t_i, t_j \mid (c_i.t_i, c_j.t_j) \in threadmap \wedge (c_i.r, c_i.t_i) \in \mathcal{T} \wedge (c_j.s, c_j.t_j) \in \mathcal{T}} \ Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)
\end{aligned}
$$

The correct versions of $SoG(c_i.r)$, $Conn(c_i.r, c_i.t_i, c_j.s, c_j.t_j)$, and $SiG(c_j.s)$ must be used for each interaction between a pair of pins.

5 Compositional Minimization

The previous section presented simple definitions of composition. The general form has been to combine the behavior specifications for the interacting components with the behavior specifications for the glue representing different types of interaction. This general form results in models that are typically larger than necessary. For example, often, some of a component's behavior is irrelevant to a composition, because not all sink pins are stimulated. This section presents a series of modified composition definitions. Each one attempts to eliminate unnecessary CSP process descriptions.

One common pattern in eliminating CSP process descriptions is to restrict the process description of a component to include only some of the logical thread processes of which the component is composed. The restriction of component process description P_c to a set of logical threads T, $P_c \lceil T$, is defined by transforming P_c as follows:

Iterate through each CSP process of which P_c is composed; for each process P

- If P represents a logical thread t and $t \in T$, leave P unchanged.
- If P represents a logical thread t and $t \notin T$, replace P with *NOTHING*.
- If P represents an auxiliary process and $\exists\, t \in T \bullet \alpha P \cap \alpha t \neq \emptyset$, leave P unchanged.
- If P represents an auxiliary process and $\forall\, t \in T \bullet \alpha P \cap \alpha t = \emptyset$, replace P with *NOTHING*.

All processes remain connected by their original composition operators (i.e., $\|$ or $|||$).

Next, iterate through the composition and collapse the expression by applying the following two rules:

- $P \parallel NOTHING = P$
- $P \;|||\; NOTHING = P$

The result is the value of $P_c \lceil T$.

The above technique is algorithmic for reasons of simplicity. We considered options that would replace processes with *Stop* or *Run*, but none could be applied as uniformly as the algorithmic solution. While $P \parallel Run_{\alpha P} = P$, $P \;|||\; Run_{\alpha P} = Run_{\alpha P}$. Likewise, while $P \;|||\; Stop_{\alpha P} = P$, $P \parallel Stop_{\alpha P} = Stop_{\alpha P}$. Consequently, tool support would be used to produce definitions for processes of the form $P_c \lceil T$, much as tool support would be used to produce definitions for P'_c.

5.1 Compositional Minimization: Interactions

A more compact representation can be produced using the following redefinition of $I(c_i.r, c_j.s)$:

$$I(c_i.r, c_j.s) = (P'_{c_i} \lceil T_{c_i.r}) \parallel MinGlue(c_i.r, c_j.s) \parallel (P'_{c_j} \lceil T_{c_j.s})$$

$T_{c_i.r}$ is the set of logical threads of c_i that can participate in interactions via source pin r. $T_{c_j.s}$ is the set of logical threads of c_j that can participate in interactions via sink pin s. $T_{c_i.r}$ and $T_{c_j.s}$ are calculated as the union of those logical threads directly involved (T_{inv}) in interactions on a pin and those logical threads interacting (T_{int}) with the logical threads directly involved. Interacting logical threads are those that can influence the behavior of the involved logical threads directly or indirectly. *interacts* is a relation specifying those logical threads that interact with each other by synchronizing on one or more internal, non-pin events. $T_{c_i.r}$ and $T_{c_j.s}$ are formally defined as

$$
\begin{aligned}
&T_{c_i.r} = T_{inv} \cup T_{int}, \text{ where}\\
&\quad T_{inv} = \{t_i \mid (c_i.r, c_i.t_i) \in \mathcal{T} \wedge \exists\, t_j \bullet (c_j.s, c_j.t_j) \in \mathcal{T} \wedge (c_i.t_i, c_j.t_j) \in \mathit{threadmap}\}\\
&\quad T_{int} = \{t_i \in (T_{all} \setminus T_{inv}) \mid \exists\, t_j \in T_{inv} \bullet (t_i, t_j) \in \mathit{interacts}^{+}\}\\
&\quad T_{all} = \textstyle\bigcup_{s_i \in c_i.\mathcal{S}} \{t \mid (c_i.s_i, c_i.t) \in \mathcal{T}\}\\
&\quad \mathit{interacts} = \{t_i, t_j \in T_{all} \mid t_i \neq t_j \wedge \exists\, e \bullet e \in (\alpha t_i \cap \alpha t_j) \wedge e \notin (c_i.\mathcal{R} \cup c_i.\mathcal{S})\}
\end{aligned}
$$

$$
\begin{aligned}
&T_{c_j.s} = T_{inv} \cup T_{int}, \text{ where}\\
&\quad T_{inv} = \{t_j \mid (c_j.s, c_j.t_j) \in \mathcal{T} \wedge \exists\, t_i \bullet (c_i.r, c_i.t_i) \in \mathcal{T} \wedge (c_i.t_i, c_j.t_j) \in \mathit{threadmap}\}\\
&\quad T_{int} = \{t_i \in (T_{all} \setminus T_{inv}) \mid \exists\, t_j \in T_{inv} \bullet (t_i, t_j) \in \mathit{interacts}^{+}\}\\
&\quad T_{all} = \textstyle\bigcup_{s_i \in c_s.\mathcal{S}} \{t \mid (c_s, s_i, t) \in \mathcal{T}\}\\
&\quad \mathit{interacts} = \{t_i, t_j \in T_{all} \mid t_i \neq t_j \wedge \exists\, e \bullet e \in (\alpha t_i \cap \alpha t_j) \wedge e \notin (c_j.\mathcal{R} \cup c_j.\mathcal{S})\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{MinGlue}(c_i.r, c_j.s) = {} & (|||_{t_i \mid (c_i.r, c_i.t_i) \in \mathcal{T} \wedge \exists\, t_j \bullet (c_j.s, c_j.t_j) \in \mathcal{T} \wedge (c_i.t_i, c_j.t_j) \in \mathit{threadmap}}\ \mathit{SoG}(c_i.r, c_i.t_i)) \parallel\\
& (|||_{t_i, t_j \mid (c_i.r, c_i.t_i) \in \mathcal{T} \wedge (c_j.s, c_j.t_j) \in \mathcal{T} \wedge (c_i.t_i, c_j.t_j) \in \mathit{threadmap}}\ \mathit{Conn}(c_i.r, c_i.t_i, c_j.s, c_j.t_j)) \parallel\\
& (|||_{t_j \mid (c_j.s, c_j.t_j) \in \mathcal{T} \wedge \exists\, t_i \bullet (c_i.r, c_i.t_i) \in \mathcal{T} \wedge (c_i.t_i, c_j.t_j) \in \mathit{threadmap}}\ \mathit{SiG}(c_j.s, c_j.t_j))
\end{aligned}
$$

Note that the specifications of T_{int} above are not perfect. The form specified is correct in that it will not leave out logical threads that could interact with the involved logical threads. But, it may include logical threads that do not interact with the involved logical threads. Ideally, a logical thread should be included in T_{int} only if it contains an event in its alphabet that is also used by a member of T_{inv} and the two logical threads are composed using a $\parallel$ operator. Logical threads composed using the $|||$ operator cannot, by definition, interact, and therefore should not be retained.

5.2 Compositional Minimization: Reentrant Interactions

Since reentrant interactions always map logical threads of the source pin's component to logical threads of the sink pin's component in a one-to-one manner, we can actually do away with the sink and source glues. All we need is a simple process connecting a source pin directly to a sink pin, for example

$$
\begin{aligned}
&\mathit{SoG}(c_i.r) = \mathit{NOTHING}\\
&\mathit{SoG}(c_i.r, c_i.t_i) = \mathit{NOTHING}\\
&\mathit{Conn}(c_i.r, c_i.t_i, c_j.s, c_j.t_j) = c_i.t_i.r \rightarrow c_j.t_j.s \rightarrow c_j.t_j.s \rightarrow c_i.t_i.r \rightarrow \mathit{Conn}(c_i.r, c_i.t_i, c_j.s, c_j.t_j)\\
&\mathit{SiG}(c_j.s) = \mathit{NOTHING}\\
&\mathit{SiG}(c_j.s, c_j.t_j) = \mathit{NOTHING}
\end{aligned}
$$

As when producing a restriction (e.g., $P'_c \upharpoonleft T$), the resulting CSP expression (an interaction composition or assembly composition) should be reduced using the following rules:

$$\begin{aligned}
&P \parallel \mathit{NOTHING} = P \\
&P \mid\mid\mid \mathit{NOTHING} = P
\end{aligned}$$

5.3 Compositional Minimization: Assemblies

The simple composition semantics for assemblies given earlier can include more information than is necessary. For example, sink glues would be included for sinks that are not stimulated. We can certainly figure out which sinks are not connected to any sources and exclude their behavior, but we can actually take this idea a bit further. We should also take into account which sinks can be stimulated by the assembly's environment.

We use the set *EnvSinks* to represent which logical threads of which sink pins the environment is allowed to stimulate. *EnvSinks* is a set of (sink pin, logical thread) pairs. The value of *EnvSinks* is supplied by the component assembler, but must conform to the constraint: $\mathit{EnvSinks} \subset \mathcal{T}$. That is, every pair is well defined, also appearing in $\mathcal{T}$, the set of allocated logical threads of the assembly.

Given this information, we can determine which logical threads can be stimulated directly or indirectly within that environment. The set of mappings between logical threads that can be stimulated is *usable*, which is defined as

$$\begin{aligned}
\mathit{usable} = \{(c_i.t_i, c_j.t_j) \in \mathit{threadmap} \mid \exists (c.s, c.t) \in \mathit{EnvSinks} \bullet ((c.t = c_i.t_i) \vee \\
(c.t, c_i.t_i) \in \mathit{threadmap}^+)\}
\end{aligned}$$

That is, a logical thread of a component can be stimulated only if it can be stimulated directly by the environment or is connected transitively back to a logical thread that can be stimulated directly by the environment. Only the subset of logical thread mappings from *threadmap* for which the source pin's logical thread can be stimulated is included in *usable*.

Using *EnvSinks* and *usable*, we define a more compact *Assembly* CSP process as

$$\mathit{Assembly} = \mathit{Components}(AC) \parallel \mathit{SourceGlues}(AC) \parallel \mathit{SinkGlues}(AC) \parallel \mathit{Connections}(AC)$$

$$\begin{aligned}
&\mathit{Components}(AC) = \mid\mid\mid_{c \in AC} P'_c \upharpoonleft T_c \\
&\quad T_c = T_{inv} \cup T_{int} \\
&\qquad T_{inv} = \{t \mid \exists\, c_i.t_i \bullet (c_i.t_i, c.t) \in \mathit{usable}\} \cup \{t \mid \exists\, p \bullet (c.p, c.t) \in \mathit{EnvSinks}\} \\
&\qquad T_{int} = \{t_i \in (T_{all} \setminus T_{inv}) \mid \exists\, t_j \in T_{inv} \bullet (t_i, t_j) \in \mathit{interacts}^+\} \\
&\qquad T_{all} = \bigcup_{s \in c.\mathcal{S}} \{t \mid (c.s, c.t) \in \mathcal{T}\} \\
&\qquad \mathit{interacts} = \{t_i, t_j \in T_{all} \mid t_i \neq t_j \wedge \exists\, e \bullet e \in (\alpha t_i \cap \alpha t_j) \wedge e \notin (c.\mathcal{R} \cup c.\mathcal{S})\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{SourceGlues}(AC) = \mid\mid\mid_{c.r \in \mathit{AllSources}} \mathit{SoGlue}(c.r) \\
&\quad \mathit{AllSources} = \{c.r \mid c \in AC \wedge r \in c.\mathcal{R} \wedge \exists\, c_j.s \bullet (c.r, c_j.s) \in \mathit{top}\} \\
&\quad \mathit{SoGlue}(c.r) = \mid\mid\mid_{t_i \mid (c.r, c.t_i) \in \mathcal{T} \wedge \exists\, c_j.t_j \bullet (c.t_i, c_j.t_j) \in \mathit{usable}} \mathit{SoG}(c.r, c.t_i)
\end{aligned}$$

$$
\begin{aligned}
&\mathit{SinkGlues}(AC) = |||_{c.s \in \mathit{AllSinks}}\ \mathit{SiGlue}(c.s)\\
&\qquad \mathit{AllSinks} = \{c.s \mid c \in AC \wedge s \in c.\mathcal{S} \wedge \exists\, c_i.r \bullet (c_i.r, c.s) \in \mathit{top}\}\\
&\qquad \mathit{SiGlue}(c.s) = |||_{t_j \mid (c.s, c.t_j) \in \mathcal{T} \wedge \exists\, c_i.t_i \bullet (c_i.t_i, c.t_j) \in \mathit{usable}}\ \mathit{SiG}(c.s, c.t_j)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Connections}(AC) = |||_{(c_i.r) \in \mathit{AllSources}, (c_j.s) \in \mathit{AllSinks} \mid (c_i.r, c_j.s) \in \mathit{top}}\ \mathit{Conns}(c_i.r, c_j.s)\\
&\qquad \mathit{Conns}(c_i.r, c_j.s) = |||_{t_i, t_j \mid (c_i.r, c_i.t_i) \in \mathcal{T} \wedge (c_j.s, c_j.t_j) \in \mathcal{T} \wedge (c_i.t_i, c_j.t_j) \in \mathit{usable}}\ \mathit{Conn}(c_i.r, c_i.t_i, c_j.s, c_j.t_j)
\end{aligned}
$$

We have restricted the composition to include only behavior that can be stimulated, directly or indirectly, by the assembly's environment. This is accomplished using a combination of techniques:

- *usable*, the constrained version of *threadmap*, only contains mappings between logical threads that can be stimulated in a particular environment, as given by *EnvSinks*. This constraint is key to the other restrictions.
- The restriction $P'_c \lceil T_c$ removes logical threads that cannot be stimulated from components' behavior descriptions.
- The use of *usable* in the quantified $|||$ operators in the definitions of $\mathit{SoGlue}(c_i.r)$, $\mathit{SiGlue}(c_j.s)$, and $\mathit{Conn}(c_i.r, c_j.s)$ prevents glues and connections from being included for logical threads that cannot be stimulated when the assembly is used in a particular environment.

5.4 Compositional Minimization: Further Optimizations

The previous sections outlined some simple state-space minimization techniques to illustrate possibilities. We are well aware that many other techniques are likely applicable; this is an area we continue to explore.

There are also more opportunities to simplify the CSP we have defined for interaction semantics. For example, we are considering whether the *Conn* process interposed between source and sink glues is really necessary. If we can handle the binding of sources' logical threads to sinks' logical threads in sink glue processes (*SiG*s), we should be able to remove *Conn* and reduce the size of generated state spaces a bit more.

Further state-space reduction would be possible when we employ a model checker to verify such descriptions. In particular, we would like to use well-proven techniques such as partial-order reduction, assume-guarantee reasoning, and efficient state-space representations such as ordered binary decision diagrams (OBDDs) for verifying assembly specifications.

6 Examples

The following sections present some simple examples to illustrate the semantics of CL that have been presented.

6.1 Example of Simple Reentrant Interaction

Component c_1 has two sinks, x and y; both of which are mutex sinks. It has one source, r, which is a synchronous source. $c_1.\mathcal{S} = \{x, y\}$. $c_1.\mathcal{R} = \{r\}$.

Component c_2 has one sink, s, which is a reentrant sink. It has one source, z, which is a synchronous source. $c_2.\mathcal{S} = \{s\}$. $c_2.\mathcal{R} = \{z\}$.

Source r is connected to sink s, meaning that $top = \{(c_1.r, c_2.s)\}$. A diagram showing this assembly is shown in Figure 10.

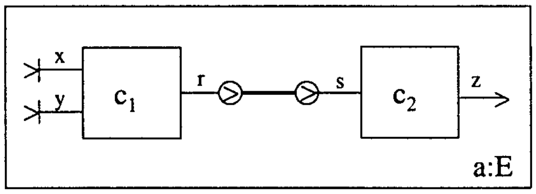


Figure 10: Reentrant Interaction $c_1.r \rightsquigarrow c_2.s$

c_1 has two reactions, one for each mutex sink. P_{c_1} is defined in terms of these reactions, as follows:

$$
\begin{aligned}
&P_{c_1} = R_x \mid\mid\mid R_y \\
&R_x = x \rightarrow r \rightarrow r \rightarrow x \rightarrow R_x \\
&R_y = y \rightarrow r \rightarrow r \rightarrow y \rightarrow R_y
\end{aligned}
$$

The transformed version P'_{c_1} is easy to produce in this case. There are no reentrant sinks, so each reaction produces one logical thread. Specifically, R_x produces logical thread t_1, and R_y produces t_2. Consequently, $\mathcal{T}$ for c_1 only is equal to $\{(c_1.x, c_1.t_1), (c_1.r, c_1.t_1), (c_1.y, c_1.t_2), (c_1.r, c_1.t_2)\}$. P'_{c_1} is defined as

$$
\begin{aligned}
&P'_{c_1} = LT_{t_1} \mid\mid\mid LT_{t_2} \\
&LT_{t_1} = R_x\,[\forall p \in \{x, y, r\} \bullet c_1.t_1.p \backslash p] \\
&LT_{t_2} = R_y\,[\forall p \in \{x, y, r\} \bullet c_1.t_2.p \backslash p]
\end{aligned}
$$

c_2 only has one reaction. P_{c_2} is defined in terms of this reaction, as follows:

$$
\begin{aligned}
&P_{c_2} = R_s \\
&R_s = s \rightarrow z \rightarrow z \rightarrow s \rightarrow R_s
\end{aligned}
$$

However, in the context of this interaction, c_2 has two logical threads (t_3 and t_4), one for each logical thread in c_2's environment that can interact with s. Each logical thread is a copy of R_s. $\mathcal{T}$ for c_2 only is equal to $\{(c_2.s, c_2.t_3), (c_2.z, c_2.t_3), (c_2.s, c_2.t_4), (c_2.z, c_2.t_4)\}$. The transformed version P'_{c_2} is defined as

$$P'_{c_2} = |||_{t \mid (c_2,s,t) \in \mathcal{T}}\ LT_t$$
$$LT_t = R_s\,[\forall p \in \{s, z\} \bullet c_2.t.p \backslash p]$$

The full definition of $\mathcal{T}$ is equal to
$\{(c_1.x, c_1.t_1), (c_1.r, c_1.t_1), (c_1.y, c_1.t_2), (c_1.r, c_1.t_2), (c_2.s, c_2.t_3),$
$(c_2.z, c_2.t_3), (c_2.s, c_2.t_4), (c_2.z, c_2.t_4)\}$.

There are two possible definitions of *threadmap* that satisfy the constraints identified earlier, and the choice of allowable values is arbitrary. For this example, we define
$threadmap = \{(c_1.t_1, c_2.t_3), (c_1.t_2, c_2.t_4)\}$.

After filling in values particular to this interaction, $Glue(c_1.r, c_2.s)$ is

$$Glue(c_1.r, c_2.s) = SoG(c_1.r) \parallel (Conn(c_1.r, c_1.t_1, c_2.s, c_2.t_3) \;|||\; Conn(c_1.r, c_1.t_2, c_2.s, c_2.t_4)) \parallel SiG(c_2.s)$$

The composed behavior of the interaction between r and s is

$$I(c_1.r, c_2.s) = P'_{c_1} \parallel Glue(c_1.r, c_2.s) \parallel P'_{c_2}$$

Finally, below we re-present all of the CSP processes used in defining the interaction between $c_1.r$ and $c_2.s$ after all substitutions and renaming have been performed:

$$I(c_1.r, c_2.s) = P'_{c_1} \parallel Glue(c_1.r, c_2.s) \parallel P'_{c_2}$$

$$P'_{c_1} = LT_{t_1} \;|||\; LT_{t_2}$$
$$LT_{t_1} = c_1.t_1.x \rightarrow c_1.t_1.r \rightarrow c_1.t_1.r \rightarrow c_1.t_1.x \rightarrow LT_{t_1}$$
$$LT_{t_2} = c_1.t_2.y \rightarrow c_1.t_2.r \rightarrow c_1.t_2.r \rightarrow c_1.t_2.y \rightarrow LT_{t_2}$$

$$P'_{c_2} = LT_{t_3} \;|||\; LT_{t_4}$$
$$LT_{t_3} = c_2.t_3.s \rightarrow c_2.t_3.z \rightarrow c_2.t_3.z \rightarrow c_2.t_3.s \rightarrow LT_{t_3}$$
$$LT_{t_4} = c_2.t_4.s \rightarrow c_2.t_4.z \rightarrow c_2.t_4.z \rightarrow c_2.t_4.s \rightarrow LT_{t_4}$$

$$Glue(c_1.r, c_2.s) = SoG(c_1.r) \parallel (Conn(c_1.r, c_1.t_1, c_2.s, c_2.t_3) \;|||\; Conn(c_1.r, c_1.t_2, c_2.s, c_2.t_4)) \parallel SiG(c_2.s)$$

$$SoG(c_1.r) = SoG(c_1.r, c_1.t_1) \;|||\; SoG(c_1.r, c_1.t_2)$$
$$SoG(c_1.r, c_1.t_1) = c_1.t_1.r \rightarrow c_1.t_1.r.left \rightarrow c_1.t_1.r.left \rightarrow c_1.t_1.r \rightarrow SoG(c_1.r, c_1.t_1)$$
$$SoG(c_1.r, c_1.t_2) = c_1.t_2.r \rightarrow c_1.t_2.r.left \rightarrow c_1.t_2.r.left \rightarrow c_1.t_2.r \rightarrow SoG(c_1.r, c_1.t_2)$$

$$Conn(c_1.r, c_1.t_1, c_2.s, c_2.t_3) = c_1.t_1.r.left \rightarrow c_2.t_3.s.right \rightarrow c_2.t_3.s.right \rightarrow c_1.t_1.r.left \rightarrow Conn(c_1.r, c_1.t_1, c_2.s, c_2.t_3)$$
$$Conn(c_1.r, c_1.t_2, c_2.s, c_2.t_4) = c_1.t_2.r.left \rightarrow c_2.t_4.s.right \rightarrow c_2.t_4.s.right \rightarrow c_1.t_2.r.left \rightarrow Conn(c_1.r, c_1.t_2, c_2.s, c_2.t_4)$$

$$
\begin{aligned}
&SiG(c_2.s) = SiG(c_2.s, c_2.t_3) \mathbin{|||} SiG(c_2.s, c_2.t_4)\\
&SiG(c_2.s, c_2.t_3) = c_2.t_3.s.right \rightarrow c_2.t_3.s \rightarrow c_2.t_3.s \rightarrow c_2.t_3.s.right \rightarrow SiG(c_2.s, c_2.t_3)\\
&SiG(c_2.s, c_2.t_4) = c_2.t_4.s.right \rightarrow c_2.t_4.s \rightarrow c_2.t_4.s \rightarrow c_2.t_4.s.right \rightarrow SiG(c_2.s, c_2.t_4)
\end{aligned}
$$

6.2 Example of Simple Mutex Interaction

Component c_1 has two sinks, x and y, both of which are mutex sinks. It has one source, r, which is a synchronous source. $c_1.\mathcal{S} = \{x, y\}$. $c_1.\mathcal{R} = \{r\}$.

Component c_2 has one sink, s, which is a mutex sink. It has one source, z, which is a synchronous source. $c_2.\mathcal{S} = \{s\}$. $c_2.\mathcal{R} = \{z\}$.

Source r is connected to sink s, meaning that $top = \{(c_1.r, c_2.s)\}$. A diagram showing this assembly is shown in Figure 11.

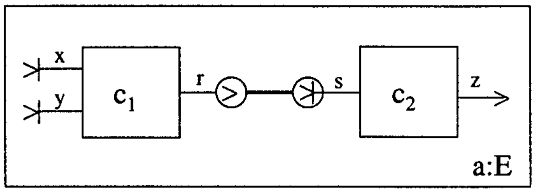


Figure 11: Mutex Interaction $c_1.r \rightsquigarrow c_2.s$

c_1 has two reactions, one for each mutex sink. P_{c_1} is defined in terms of these reactions, as follows:

$$
\begin{aligned}
&P_{c_1} = R_x \mathbin{|||} R_y\\
&R_x = x \rightarrow r \rightarrow r \rightarrow x \rightarrow R_x\\
&R_y = y \rightarrow r \rightarrow r \rightarrow y \rightarrow R_y
\end{aligned}
$$

The transformed version P'_{c_1} is easy to produce; since there are no reentrant sink pins, each reaction produces one logical thread. Specifically, R_x produces logical thread t_1, and R_y produces t_2. Consequently, $\mathcal{T}$ for c_1 only is equal to $\{(c_1.x, c_1.t_1), (c_1.r, c_1.t_1), (c_1.y, c_1.t_2), (c_1.r, c_1.t_2)\}$. P'_{c_1} is defined as

$$
\begin{aligned}
&P'_{c_1} = LT_{t_1} \mathbin{|||} LT_{t_2}\\
&LT_{t_1} = R_x\,[\forall p \in \{x, y, r\} \bullet c_1.t_1.p \backslash p]\\
&LT_{t_2} = R_y\,[\forall p \in \{x, y, r\} \bullet c_1.t_2.p \backslash p]
\end{aligned}
$$

c_2 has only one reaction, that for its mutex sink. P_{c_2} is defined in terms of this reaction as follows:

$$
\begin{aligned}
&P_{c_2} = R_s\\
&R_s = s \rightarrow z \rightarrow z \rightarrow s \rightarrow R_s
\end{aligned}
$$

The transformed version P'_{c_2} is also easy to produce; since its sink is not a reentrant sink, its reaction produces one logical thread, t_3. $\mathcal{T}$ for c_2 only is equal to $\{(c_2.s, c_2.t_3), (c_2.z, c_2.t_3)\}$. P'_{c_2} is defined as

$$P'_{c_2} = LT_{t_3}$$
$$LT_{t_3} = R_s\,[\forall p \in \{s, z\} \bullet c_2.t_3.p \backslash p]$$

The full definition of $\mathcal{T}$ is equal to $\{(c_1.x, c_1.t_1),\ (c_1.r, c_1.t_1),\ (c_1.y, c_1.t_2),\ (c_1.r, c_1.t_2),\ (c_2.s, c_2.t_3),\ (c_2.z, c_2.t_3)\}$.

There is only one definition of *threadmap* that satisfies the constraints identified earlier: $\mathit{threadmap} = \{(c_1.t_1, c_2.t_3),\ (c_1.t_2, c_2.t_3)\}$.

After filling in values particular to this interaction, $\mathit{Glue}(c_1.r, c_2.s)$ is

$$\begin{aligned}\mathit{Glue}(c_1.r, c_2.s) = {} & \mathit{SoG}(c_1.r) \parallel (\mathit{Conn}(c_1.r, c_1.t_1, c_2.s, c_2.t_3) \mid\mid\mid \mathit{Conn}(c_1.r, c_1.t_2, c_2.s, c_2.t_3)) \parallel \\ & \mathit{SiG}(c_2.s)\end{aligned}$$

The composed behavior of the interaction between r and s is

$$I(c_1.r, c_2.s) = P'_{c_1} \parallel \mathit{Glue}(c_1.r, c_2.s) \parallel P'_{c_2}$$

Finally, below we re-present all of the CSP processes used in defining the interaction between r and s after all substitutions and renaming have been performed:

$$I(c_1.r, c_2.s) = P'_{c_1} \parallel \mathit{Glue}(c_1.r, c_2.s) \parallel P'_{c_2}$$

$$\begin{aligned}P'_{c_1} &= LT_{t_1} \mid\mid\mid LT_{t_2}\\ LT_{t_1} &= c_1.t_1.x \to c_1.t_1.r \to c_1.t_1.r \to c_1.t_1.x \to LT_{t_1}\\ LT_{t_2} &= c_1.t_2.y \to c_1.t_2.r \to c_1.t_2.r \to c_1.t_2.y \to LT_{t_2}\end{aligned}$$

$$\begin{aligned}P'_{c_2} &= LT_{t_3}\\ LT_{t_3} &= c_2.t_3.s \to c_2.t_3.z \to c_2.t_3.z \to c_2.t_3.s \to LT_{t_3}\end{aligned}$$

$$\begin{aligned}\mathit{Glue}(c_1.r, c_2.s) = {} & \mathit{SoG}(c_1.r) \parallel (\mathit{Conn}(c_1.r, c_1.t_1, c_2.s, c_2.t_3) \mid\mid\mid \mathit{Conn}(c_1.r, c_1.t_2, c_2.s, c_2.t_3)) \parallel \\ & \mathit{SiG}(c_2.s)\end{aligned}$$

$$\begin{aligned}\mathit{SoG}(c_1.r) &= \mathit{SoG}(c_1.r, c_1.t_1) \mid\mid\mid \mathit{SoG}(c_1.r, c_1.t_2)\\ \mathit{SoG}(c_1.r, c_1.t_1) &= c_1.t_1.r \to c_1.t_1.r.\mathit{left} \to c_1.t_1.r.\mathit{left} \to c_1.t_1.r \to \mathit{SoG}(c_1.r, c_1.t_1)\\ \mathit{SoG}(c_1.r, c_1.t_2) &= c_1.t_2.r \to c_1.t_2.r.\mathit{left} \to c_1.t_2.r.\mathit{left} \to c_1.t_2.r \to \mathit{SoG}(c_1.r, c_1.t_2)\end{aligned}$$

$$\begin{aligned}\mathit{Conn}(c_1.r, c_1.t_1, c_2.s, c_2.t_3) = {} & c_1.t_1.r.\mathit{left} \to c_1.t_1.r.\mathit{right} \to c_1.t_1.r.\mathit{right} \to c_1.t_1.r.\mathit{left} \to \\ & \mathit{Conn}(c_1.r, c_1.t_1, c_2.s, c_2.t_3)\\ \mathit{Conn}(c_1.r, c_1.t_2, c_2.s, c_2.t_3) = {} & c_1.t_2.r.\mathit{left} \to c_1.t_2.r.\mathit{right} \to c_1.t_2.r.\mathit{right} \to c_1.t_2.r.\mathit{left} \to \\ & \mathit{Conn}(c_1.r, c_1.t_2, c_2.s, c_2.t_3)\end{aligned}$$

$$\begin{aligned}&\mathit{SiG}(c_2.s) = \mathit{SiG}(c_2.s, c_2.t_3)\\ &\mathit{SiG}(c_2.s, c_2.t_3) = \mathit{WaitRight}(c_2.s, c_2.t_3)_{<>}\\ &\quad \mathit{WaitRight}(c_2.s, c_2.t_3)_{<>} = \Box_{c_i.r \in \{c_1.r\}}\, c_i?t_i!r!\mathit{right} \to c_2.t_3.s \to\\ &\qquad\qquad \mathit{WaitRight}(c_2.s, c_2.t_3)_{<c_i.t_i.r>}\\ &\quad \mathit{WaitRight}(c_2.s, c_2.t_3)_{Q^\frown <x.y.z>} = (\Box_{c_i.r \in \{c_1.r\}}\, c_i?t_i!r!\mathit{right} \to\\ &\qquad\qquad \mathit{WaitRight}(c_2.s, c_2.t_3)_{<c_i.t_i.r>^\frown Q^\frown <x.y.z>})\end{aligned}$$

$$\begin{array}{ll} & \square c_2.t_3.s \rightarrow x.y.z.right \rightarrow Next(c_2.s, c_2.t_3)_Q \\ Next(c_2.s, c_2.t_3)_{<>} & = WaitRight(c_2.s, c_2.t_3)_{<>} \\ Next(c_2.s, c_2.t_3)_{Q^\frown <x.y.z>} & = c_2.t_3.s \rightarrow WaitRight(c_2.s, c_2.t_3)_{Q^\frown <x.y.z>} \end{array}$$

6.3 Example of Simple Assembly

The example, shown in Figure 12, presents the modeling of a real-life scenario in Pin. The software assembly consists of two GUIs, G_1 and G_2, operating on different client machines that need the services of database server D, statistical library S, and printer P. In order to complete its job, a GUI component gets data from D (e.g., a list of a firm's employees and their salaries or stock indexes over a period), uses functions from S to process it (mean, correlation), and sends the results to printer P.

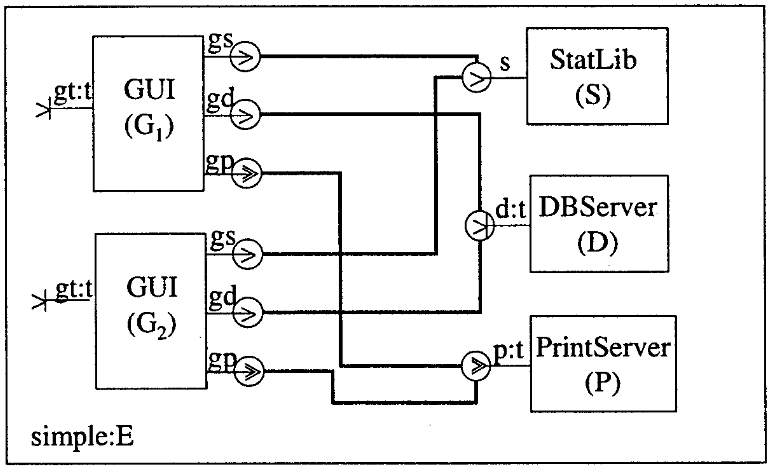


Figure 12: Simple Assembly

G_1 and G_2 represent GUI components, D represents the Database server, S represents the statistical library component, and P represents the printer component.

The component specifications are

$P_{G_1} = R_g$
$P_{G_2} = R_g$
$R_g = gt \rightarrow gd \rightarrow gd \rightarrow gs \rightarrow gs \rightarrow gp \rightarrow gt \rightarrow R_g$

$P_D = R_d$
$R_d = d \rightarrow p_{dint} \rightarrow d \rightarrow R_d$ (where p_{dint} corresponds to internal processing)

$P_S = R_s$
$R_s = s \rightarrow p_{sint} \rightarrow s \rightarrow R_s$ (where p_{sint} corresponds to internal processing)

$P_P = R_p$
$R_p = p \rightarrow p_{pint} \rightarrow R_p$ (where p_{pint} corresponds to internal processing)

$$\mathcal{T} = \{(G_1.gt, G_1.t_1), (G_1.gs, G_1.t_1), (G_1.gd, G_1.t_1), (G_1.gp, G_1.t_1), (G_2.gt, G_2.t_2), (G_2.gs, G_2.t_2), (G_2.gd, G_2.t_2), (G_2.gp, G_2.t_2), (D.d, D.t_3), (P.p, P.t_4), (S.s, S.t_5), (S.s, S.t_6)\}$$

The transformed processes are

$$P'_{G_1} = LT_{t_1}$$
$$LT_{t_1} = R_g[\forall\, p \in \{gt, gd, gs, gp\} \bullet G_1.t_1.p \backslash p]$$

$$P'_{G_2} = LT_{t_2}$$
$$LT_{t_2} = R_g[\forall\, p \in \{gt, gd, gs, gp\} \bullet G_2.t_2.p \backslash p]$$

$$P'_D = LT_{t_3}$$
$$LT_{t_3} = R_d[\forall\, p \in \{d\} \bullet D.t_3.p \backslash p, \forall\, e \in \{p_{dint}\} \bullet D.e \backslash e]$$

$$P'_P = LT_{t_4}$$
$$LT_{t_4} = R_p[\forall\, q \in \{p\} \bullet P.t_4.q \backslash q, \forall\, e \in \{p_{pint}\} \bullet P.e \backslash e]$$

$$P'_S = LT_{t_5} \;|||\; LT_{t_6}$$
$$LT_{t_5} = R_s[\forall\, p \in \{s\} \bullet S.t_5.p \backslash p, \forall\, e \in \{p_{sint}\} \bullet S.e \backslash e]$$
$$LT_{t_6} = R_s[\forall\, p \in \{s\} \bullet S.t_6.p \backslash p, \forall\, e \in \{p_{sint}\} \bullet S.e \backslash e]$$

$$threadmap = \{(G_1.t_1, D.t_3), (G_2.t_2, D.t_3), (G_1.t_1, P.t_4), (G_2.t_2, P.t_4), (G_1.t_1, S.t_5), (G_2.t_2, S.t_6)\}$$
$$AC = \{G_1, G_2, D, S, P\}$$
$$Assembly = Components(AC) \parallel SourceGlues(AC) \parallel SinkGlues(AC) \parallel Connections(AC)$$

$$Components(AC) = P'_{G_1} \;|||\; P'_{G_2} \;|||\; P'_D \;|||\; P'_S \;|||\; P'_P$$

$$SourceGlues(AC) = SoG(G_1.gd) \;|||\; SoG(G_1.gs) \;|||\; SoG(G_1.gp) \;|||\; SoG(G_2.gd) \;|||\; SoG(G_2.gs) \;|||\; SoG(G_2.gp)$$

$SoG(G_1.gd) = SoG(G_1.gd, G_1.t_1)$	using the mutex SoG
$SoG(G_1.gs) = SoG(G_1.gs, G_1.t_1)$	using the reentrant SoG
$SoG(G_1.gp) = SoG(G_1.gp, G_1.t_1)$	using the asynchronous SoG
$SoG(G_2.gd) = SoG(G_2.gd, G_2.t_2)$	using the mutex SoG
$SoG(G_2.gs) = SoG(G_2.gs, G_2.t_2)$	using the reentrant SoG
$SoG(G_2.gp) = SoG(G_2.gp, G_2.t_2)$	using the asynchronous SoG

$$SinkGlues(AC) = SiG(G_1.gt) \;|||\; SiG(G_2.gt) \;|||\; SiG(D.d) \;|||\; SiG(S.s) \;|||\; SiG(P.p)$$

$SiG(G_1.gt) = SiG(G_1.gt, G_1.t_1)$	using the mutex SiG
$SiG(G_2.gt) = SiG(G_2.gt, G_2.t_2)$	using the mutex SiG
$SiG(D.d) = SiG(D.d, D.t_3)$	using the mutex SiG
$SiG(S.s) = SiG(S.s, S.t_5) \;|||\; SiG(S.s, S.t_6)$	using the reentrant SiG
$SiG(P.p) = SiG(P.p, P.t_4)$	using the asynchronous SiG

$$Connections(AC) = Conns(G_1.gd, D.d) \;|||\; Conns(G_2.gd, D.d) \;|||\; Conns(G_1.gs, S.s) \;|||\; Conns(G_2.gs, S.s) \;|||\; Conns(G_1.gp, P.p) \;|||\; Conns(G_2.gp, P.p)$$

$Conns(G_1.gd, D.d) = Conn(G_1.gd, G_1.t_1, D.d, D.t_3)$	using the mutex $Conn$

$Conns(G_2.gd, D.d) = Conn(G_2.gd, G_2.t_2, D.d, D.t_3)$	using the mutex *Conn*
$Conns(G_1.gs, S.s) = Conn(G_1.gs, G_1.t_1, S.s, S.t_5)$	using the reentrant *Conn*
$Conns(G_2.gs, S.s) = Conn(G_2.gs, G_2.t_2, S.s, S.t_6)$	using the reentrant *Conn*
$Conns(G_1.gp, P.p) = Conn(G_1.gp, G_1.t_1, P.p, P.t_4)$	using the asynchronous *Conn*
$Conns(G_2.gp, P.p) = Conn(G_2.gp, G_2.t_2, P.p, P.t_4)$	using the asynchronous *Conn*

7 Open Issues

There are a number of topics we need to address before completing CL. We are currently working on refining our representation of pin events, describing environments, defining the relation between assemblies and environments, hiding internal behavior, and defining all the details of the CL language.

7.1 Representing Pin Events

Currently, activity on a pin is denoted by a CSP event of the same name, and each interaction on a pin is denoted by a pair of such events, one for the initiation of the interaction and one for the completion of the interaction. Unfortunately, this simple approach leads to difficulty in correctly expressing some behaviors because the events are ambiguous. Ideally, initiation and completion of an interaction on a pin should be distinct events.

We are considering options for distinguishing these events. One option, similar to that used in CCS [Milner 89] and Wright [Allen 97], is to use event pairs such as e and $\overline{e}$. However, while CCS and Wright use event pairs to differentiate initiating and observing a synchronized event, we want to distinguish between initiating and completing an activity, as represented by a pair of related events. Since this is a different kind of distinction, we are considering other, similar syntactic conventions to avoid confusing anyone familiar with those languages.

Such a distinction is not just a syntactic aid to the reader. Some of our glue definitions, such as *SiG* for mutex interactions, are subtly incorrect in their current form, because we cannot distinguish initiation from completion events. There are alternative ways to correct these definitions, but making the proper distinction on pin events is the simplest option and also improves the readability of CL specifications.

7.2 Describing Environments

In Section 3, we mentioned that assemblies are defined in the context of some environment and that this environment affects the behavior of the assembly. An environment defines the semantics of the connectors used in the assembly and provides services (such as a clock) to the assembly.

An environment is also the boundary through which components interact with things not in the assembly. In the current CSP semantics, this aspect of an environment is not well specified. To analyze an assembly, we should define how components interact with the environment of their assembly.

In particular, four questions stand out:

- Which sinks are exposed to the environment? For those sink that are not exposed and that do not interact with sources within the assembly, we should exclude their behavior from the analysis model. One solution to prevent behavior from unconnected sink *s* would be to place the assembly process in || with $Stop_s$.

- What is the potential concurrency in the environment with respect to an exposed sink? Can multiple threads in the environment interact with the same sink concurrently? We have sketched a solution to this concern in Section 5.3 with *EnvSinks*, but we have no *threadmap* equivalent.
- What types of interactions exist between an environment and sinks? Are they the same types of interactions that exist within an assembly? If so, we need to model them explicitly. In the current minimized semantics for assemblies, we do not include sink glue processes for sinks connected only to the environment; this may change.
- What can be assumed regarding how an environment will behave? Is the behavior of an environment unconstrained? If so, we could model that with an explicit *Run* process, or we could leave the environment unspecified as it is now. Is the environment required to follow some protocol? If so, and if we want to gather analysis results only for scenarios in which the environment correctly follows the protocol, the environment's behavior must be given an explicit CSP definition.

The next steps will be to better define what we mean by an environment and to provide a scheme for formalizing the semantics of environments.

7.3 Relating Assemblies and Environments

Assemblies, runtime environments (containers), environment types, and deployable assemblies are interrelated terms whose meanings are still under consideration. At one point in this report, we state that an assembly is a scope for components. Likewise, an assembly is associated with an environment type. But are these really the correct distinctions? What happens in a case in which subassemblies share the environment type of an encapsulating assembly? How many runtime environments are there in such a case? Can the subassemblies be deployed independently?

We are investigating these questions. Some initial thoughts indicate that we may not have all the concepts separated correctly. For example, each assembly must be associated with an environment type, as that dictates the interaction semantics within the assembly. However, this may not imply that there is a unique runtime environment or container associated with each assembly. In a hierarchy of assemblies of the same environment type, only one runtime environment of that environment type may exist. Clearly, any assembly that does not have its own runtime environment cannot be deployed independently, and so not all assemblies are independently deployable.

7.4 Hiding Internal Behavior

Intuitively, hiding corresponds to abstraction—ignoring certain details and focusing on the rest of a model. In CSP, events can be hidden from a process, resulting in a new process with a smaller alphabet. All hidden events are replaced by τ's, or internal transitions, often resulting in a model whose state space can be further minimized.

Hiding has a natural role in composition. Why not hide all the internal activity of an assembly and focus on its externally visible behavior? We can hide all events for pins that are not exposed outside of an assembly and get the CSP equivalent. The only question, however, is what breaks if we do this? By introducing non-determinism (a natural consequence of hiding), will we be producing analysis results that are misleading?

The next steps will be to examine the consequences of applying hiding to assemblies.

7.5 Defining Language Details

Throughout this report we have shown excerpts of CL specifications, but we have not put all the pieces together. We have shown pin diagrams in which different types of pins have different symbols and connections between pins are shown in a certain way. We have shown CSP specifications for reactions, interactions, and assemblies. But we have not tied them together formally.

One task that remains is to fully define CL and how these pieces (and others) fit together. We need a complete visual syntax for pin diagrams, along with rules for valid arrangements of visual elements and to specify how additional information is associated with these visual elements. For example, reactions are associated with components, and data interfaces are associated with pins. We will have a grammar for a textual representation of this information that is suitable for machine processing and that will likely include a representation of the information found in pin diagrams.

8 Future Work

The road ahead for predictable assembly is challenging, even if we limit our focus, as we do here, exclusively to CL. Beyond resolving the open issues discussed in Section 7, there are several areas of future development, some of which must be completed before we can consider CL to have a sound basis.

8.1 Specific Environment Type Definition in CL

As we have noted, the composition semantics we defined for CL is relative to abstract environment type E. A full composition semantics requires that we define environment type E' for a concrete platform. For this purpose, we will use the switch-controller runtime described in the PACC substation automation experience report.[4] This environment type uses fixed-length queues implemented in shared memory for intercomponent communication and provides a number of runtime services for real-time computation. We will then refine the semantics defined earlier with these environment-specific characteristics.

8.2 Intermediate Representation

We have defined a composition semantics in CSP, which would be nearly sufficient if our intended analysis tool was restricted to just FDR. However, our target set of analysis tools includes a number of formal analysis systems, and, in addition, empirical analysis tools for time (latency) and potentially other properties. In compiler terms, we have defined a high-level intermediate language for CL in CSP, but we must still define one (or more?) low-level intermediate languages that are suitable for machine processing.

8.3 Visual Composition Tool Set

Our emphasis on pure composition will be most beneficial when we can provide a visual tool set for composing, analyzing, and deploying component-based software. Only then can we say that we have successfully defined a composition language and technology that supports predictable assembly. The visual notation used in this report provides a starting point for this work. Considerable work remains, however, to completely define the visual and textual syntax needed to provide automated support for composition, model generation, and, in the long run, code generation.

[4] Hissam, S.; Hudak, J.; Ivers, J.; Klein, M.; Larsson, M.; Moreno, G.; Northrop, L.; Plakosh, D.; Stafford, J.; Wallnau, K.; & Wood, W. *Predictable Assembly of Substation Automation Systems: An Experience Report.* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, to be published.

Appendix A Summary of Formal Notations Used in This Report

A.1 CSP

We use the following subset of CSP in this paper:[5]

- **processes and events:** A process describes an entity that can engage in communication events. Events may be primitive, or they can have associated data (as in $e?x$ and $e!x$, representing input and output data, respectively).
- **prefixing:** A process that engages in event e and then becomes process P is denoted $e \rightarrow P$.
- **external choice:** A process that can behave like P or Q, where the choice is made by the environment, is denoted $P \ \Box \ Q$. ("Environment" refers to the other processes that interact with the process.)
- **non-deterministic choice:** A process that can behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.
- **alphabet:** The alphabet of process P is the set of events in which the process can engage and is denoted αP.
- **parallel composition:** Processes can be composed using the $\|$ operator. Parallel processes may interact by jointly (synchronously) engaging in events that lie within the intersection of their alphabets. Conversely, if event e is in αP_1 and αP_2, then P_1 can engage in the event only if P_2 can also. That is, process $P_1 \| P_2$ is one whose behavior is permitted by both P_1 and P_2.
- **interleaving:** Processes can also be composed using the $|||$ operator. Interleaved processes never interact, even on events that lie within the intersection of their alphabets.
- **sequential composition:** A process that behaves like P, until P terminates (*Skip*) and then behaves like Q, is denoted $P \ ; \ Q$.
- **renaming:** Processes can be renamed in two ways. The expression $x:P$ represents a renamed process in which every event in P is now prefixed with x. For example, if $P = a \rightarrow b \rightarrow P$, then $x:P = x.a \rightarrow x.b \rightarrow x:P$. The expression $P[x.a \backslash a, x.b \backslash b]$ represents a renamed process in which all occurrences of a are replaced with $x.a$, and all occurrences of b are replaced with $x.b$.

In process expressions $\rightarrow$ associates to the right and binds tighter than $\Box$. So $e \rightarrow f \rightarrow P \ \Box \ g \rightarrow Q$ is equivalent to $(e \rightarrow (f \rightarrow P)) \ \Box \ (g \rightarrow Q)$.

[5] Much of this CSP primer is borrowed or adapted from Allen and associates [Allen 98].

A.2 Formal Logic

Logic	
$A \wedge B$	A and B
$A \vee B$	A or B
$\neg A$	not A
$A \Rightarrow B$	A implies B
$\forall x \in S \bullet P(x)$	universal quantification ($P(x)$ is true of all x in S)
$\exists x \in S \bullet P(x)$	existential quantification ($P(x)$ is true for at least one x in S)

Sets	
$\emptyset$	empty set
$e \in S$	e is an element of S
$S \cup T$	set union
$S \cap T$	set intersection
$S \setminus T$	set difference
$\bigcup_{e \in T} S_e$	generalized union (the union of all sets S_e)
$\#S$	set cardinality
$\{x \in S \mid P(x)\}$	set comprehension (the set of all x in S for which $P(x)$ is true)

Sequences	
$<>$	empty sequence
$S \frown T$	S concatenated with T

References

[Achermann 02]	Achermann, F.; Lumpe, M.; Schneider, J.; & Nierstrasz, O. "Piccola – a Small Composition Language". *Formal Methods for Distributed Processing-A Survey of Object-Oriented Approaches*, (January 2002): 403–426. Cambridge, UK: Cambridge University Press.
[Allen 97]	Allen, R. "A Formal Approach to Software Architecture". PhD diss., Carnegie Mellon University, May 1997. CMU-CS-97-144.
[Allen 98]	Allen, R.; Garlan, D.; & Ivers, J. "Formal Modeling and Analysis of the HLA Component Integration Standard", 70–79. *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering.* New York, NY: ACM Press, November 1998.
[Cardelli 97]	Cardelli, L. "Type Systems". *The Computer Science and Engineering Handbook*, (1997): 2208–2236. Boca Raton, FL: CRC Press. ISBN 0-8493-2909-4.
[Cimatti 00]	Cimatti, A.; Clarke, E.; Giunchiglia, F.; & Roveri, M. "NuSMV: A New Symbolic Model Verifier". *International Journal on Software Tools for Technology Transition 2*, 4 (2000): 410–425.
[Corbett 99]	Corbett, J. *Bandera Intermediate Representation (BIR) Specification Version 0.6*, November 1999. <http://www.cis.ksu.edu/~dwyer/projects/nasa-2-99.ps>.
[Garlan 93]	Garlan, D. & Shaw, M. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering*, (December 1993): 1–39. Singapore: World Scientific Publishing Company.
[Garlan 97]	Garlan, D.; Monroe, R.; & Wile, D. "Acme: An Architecture Description Interchange Language", 169–183. *Proceedings of CASCON '97.* Toronto, Ontario: IBM Canada Ltd., November 1997.
[Giannakopoulou 99]	Giannakopoulou, D. "Model Checking for Concurrent Software Architectures". PhD diss., Imperial College of Science, Technology, and Medicine, University of London, England, January 1999.
[Hissam 02]	Hissam, S.; Moreno, G.; Stafford, J.; & Wallnau, K. "Packaging Predictable Assembly", 108–125. *Proceedings of the First International IFIP/ACM Working Conference on*

Component Deployment, number 2370 in LNCS. Berlin, Germany: Springer-Verlag, June 2002.

[Hoare 85] Hoare, C. A. R. *Communicating Sequential Processes.* London, UK: Prentice-Hall, 1985.

[Holzman 97] Holzman, G. "The Model Checker Spin". *IEEE Transactions on Software Engineering 23*, 5 (May 1997): 279–295.

[Luckham 95] Luckham, D. C.; Augustin, L. M.; Kenney, J. J.; Veera, J.; Bryan, D.; & Mann, W. "Specification and Analysis of System Architecture Using Rapide". *IEEE Transactions on Software Engineering 21*, 6 (April 1995): 336–355.

[Magee 93] Magee, J.; Dulay, N.; & Kramer, J. "Structuring Parallel and Distributed Programs". *Software Engineering Journal, IEEE 8*, 2 (March 1993): 73–82.

[Magee 01] Magee, J. & Kramer, J. *Concurrency: State Models & Java Programs.* West Sussex, England: Wiley Publication, 2001.

[Milner 89] Milner, R. *Communication and Concurrency.* Hemel Hempstead, UK: Prentice-Hall, 1989.

[Milner 99] Milner, R. *Communicating and Mobile Systems: The π-Calculus.* Cambridge, England: Cambridge University Press, May 1999.

[Nierstrasz 02] Nierstrasz, O.; Arévalo, G.; Ducasse, S.; Wuyts, R.; Black, A.; Müller, P.; Zeidler, C.; Genssler, T.; & van den Born, R. "A Component Model for Field Devices". *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, number 2370 in LNCS. Berlin, Germany: Springer-Verlag, June 2002.

[Papadopoulos 98] Papadopoulos, G. & Arbab, F. *Coordination Models and Languages* (SEN-R9834, ISSN 1386-396X). Amsterdam, The Netherlands: Centrum voor Wiskunde en Informatica, December 1998.

[Plakosh 99] Plakosh, D.; Smith, D.; & Wallnau, K. *Builder's Guide for Waterbeans Components* (CMU/SEI-99-TR-024, ADA373154). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999.

[Purtillo 94] Purtillo, J. "The Polylith Software Bus". *ACM Transactions on Programming and Systems 16*, 1 (January 1994): 151–174.

[Roscoe 98] Roscoe, A. *The Theory and Practice of Concurrency.* London, UK: Prentice-Hall, 1998.

[Szyperski 97] Szyperski, C. *Component Software - Beyond Object-Oriented Programming.* Harlow, England: Addison Wesley, 1997. ISBN 0-201-17888-5.

[van Ommering 02] van Ommering, R. "The Koala Component Model". *Building Reliable Component-Based Software Systems*, (July 2002): 223–236. London, England: Artech House.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** September 2002	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** A Basis for Composition Language CL		5. **FUNDING NUMBERS** F19628-00-C-0003
6. **AUTHOR(S)** James Ivers, Nishant Sinha, Kurt Wallnau		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2002-TN-026
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. **SUPPLEMENTARY NOTES**		
12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12B **DISTRIBUTION CODE**

13. **ABSTRACT** (MAXIMUM 200 WORDS)

CL is a composition language for predictable assembly from certifiable components. An application assembly process is predictable if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interaction. CL is similar to other composition languages that combine a component and connector style of description with a core compositional semantics specified in a process algebra. CL differs from these in its explicit treatment of details that are usually abstracted or ignored. For example, CL makes explicit the allocation of execution threads to component behavior; this distinguishes concurrent from sequential behavior, and leads to potentially smaller state spaces as well as more accurate behavioral descriptions.

This report describes the main concepts of CL and its rudimentary graphical syntax. This report also defines and illustrates the compositional semantics for CL using Hoare's CSP. The twin objectives of this report are to consolidate our current thinking about an ideal CL and to provide a starting point for the design of a practical and implementable CL. This report closes with a discussion of several open issues that must be resolved before this second objective can be satisfied.

14. **SUBJECT TERMS** composition language, composition semantics, predictable assembly			15. **NUMBER OF PAGES** 62
16. **PRICE CODE**			
17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL